Decoupling Application Intelligence and its Orchestration on IoT Devices

Jose Manuel Paredes del Rio

School of Science, Aalto University
Master's Programme in ICT Innovation EIT Digital Master School - Data Science

Master's Thesis
Espoo, May , 2020

Supervisors: Prof. Kary Främling
Advisor: Edgar Ramos

Aalto University
School of Science
Master's Programme in ICT Innovation EIT Digital Master School - Data Science

ABSTRACT OF MASTER'S THESIS

Author: Jose Manuel Paredes del Rio

Title:
Decoupling Application Intelligence and its Orchestration on IoT Devices

Date: May , 2020 **Pages:** 73

Major: EIT Data Science **Code:** SCI3095

Supervisors: Prof. Kary Främling

Advisor: Edgar Ramos

The current methodology for developing applications using machine learning in IoT devices presents serious issues of scalability due to the tight coupling between Intelligence (machine learning) Services on the one hand, and applications on the other hand. The main issue is the lack of life cycle management features increasing the required efforts for deploying new intelligence in devices. This thesis aims to solve this problem by developing an Intelligence Layer that enables decoupling the Intelligence Services from applications.

This thesis describes the Intelligence Layer idea from the point of view of how intelligence orchestration should be performed and how to implement the intelligent functionality, taking into account communication protocols, security, availability and authorization. This research focuses on the IoT area and it introduces a local intelligence orchestration prototype. This prototype relies on technologies such as Open Neural Network Exchange (ONNX) to implement the Intelligent components, Inter-Process Communication (IPC) and Android as the desired platform.

The main conclusion of the research is that the Intelligence decoupling can be achieved by the proposed Intelligence Layer. Nevertheless, the IoT scope and its applications are wide, therefore the Intelligence Layer approach should be flexible enough to adapt in order to meet various application requirements. In addition, this research is not complete due to the lack of time and available Intelligent Services to test and improve the orchestration explained on this thesis, therefore the author recommends to develop more Intelligent Services to test more cases and improve the approach.

Keywords: Machine Learning, Data Science, IoT, Intelligence Orchestration

Language: English

Acknowledgements

I would like to express my gratitude to the people who make this thesis possible, Prof. Kary Främling and the NomadicLab team, in particular Edgar and Timon for the large discussions about this innovative idea; Edgar, thanks for allowing us to become part of this project and your willingness to support ; Timon, before this thesis period we were just acquaintance, now I can call you friend, I promise I will beat you at ping-pong.

I cannot finish this section without remembering my family and my Spanish friends. Sometimes, when you are further away, you feel that people are closer to you.

Espoo, May , 2020

Jose Manuel Paredes del Rio

Abbreviations and Acronyms

IL	Intelligence Layer
DIL	Device Intelligence Layer
IoT	Internet of Things
ONNX	Open Neuronal Network Exchange
OS	Operating System
LCM	Live Cycle Management
SDK	Software Development Kit
NDK	Native Development Kit
apps	Applications
ART	Android Runtime
JVM	Java Virtual Machine
AOT	ahead-of-time
XML	Extensible Markup Language
API	Application Programming Interface
REST	Representational State Transfer
ARN	Application Not Responding
VM	Virtual Machine
WORA	write once, run anywhere

Contents

Chapter 1

Introduction

Machine Learning and Big Data are trending topics in the current state of the world due to the big potential to fit the customer likeliness. They are running in the background everywhere; even normal phone users are involved because their phones continuously send information to big cloud systems without the user's knowledge. That information can then be used to train models which in the future might be deployed in applications on the phone. Although there are some researches about the increase of the devices in the coming years raising the cloud traffic to an unsustainable situation, this thesis focuses on another important issue of the cloud paradigm. The deployment of the Intelligence costs a lot of effort to adjust the application to the Intelligence requirements as it is part of the application itself. Therefore, it is needed to research on how to avoid this situation. This thesis will research the feasibility to move the Intelligence from the application to a new layer in the device, it is based on the Intelligence Layer [1].

This thesis shows how this Intelligence Layer can be defined and implemented to solve the issue already mentioned; deploying Intelligence models on end-user applications and devices.

1.1 Problem statement and scope

This thesis focuses on IoT devices. Although IoT covers a large family of devices this thesis only includes Android devices as they are used commonly by society.

The main objective is to avoid the current cost of adjusting the application when the Intelligence changes, in other words, execute Intelligence on Android IoT devices in a way that the previous data processing and the postprocessing of the model output are performed automatically without be-

ing explicitly defined in the device application. It means a change in the paradigm in which the Intelligence inference is organized by the device instead of the applications, in other terms, this thesis intends to find solutions to the Intelligence orchestration in IoT devices. To achieve this aspiration, the Intelligence Layer paper, which this thesis is based on, proposes that the intelligent feature of the applications must be separated and orchestrated in a different layer. It means that the Intelligence could be connected to applications and could be updated or requested due to application demands by a life cycle management.

1.2 Research questions

- Is it feasible to generalize the local Intelligent services APIs to applications in a device, including discovery and self-configuration?
  - In case of positive research, what does an example API for this purpose look like (taking into account security aspects, data management, policies and processing domains) applied to an automotive application domain using an android environment?
  - In case of a negative result, present an example of implementation with the main withdraws.

1.3 Objectives of the thesis

The main goal of the thesis is the improvement of the Intelligence deployment in a production environment. This goal can be further divided as follows:

- Research about how to move the Intelligence part of the applications to an external component.
- Investigate about the communications with the external component.
  - Communication behaviour.
  - Develop an API to use the Intelligence Layer
  - Discover the Intelligence Layer with an API
- How to make this external component available and useful for all the apps.
- Develop that new layer (Intelligence Layer) in a way that can inference models with the input data preprocessing and post result processing automatically itself.

1.4 Requirements

- Android is the target platform.
- Applications do not process the input/output data. These tasks must be done by the Intelligence Layer.
- Applications can use the new layer to register and call it.
- The new layer should act according to policies and security constraints.

1.5 Assumptions

- The Intelligence is already in the device. Hence, it is not needed to obtain it in from outside the device. A machine learning model is used as Intelligence in this Thesis.

1.6 Structure of the Thesis

In order to fulfill the research objectives, this thesis first describes issues with the current paradigm in Chapter 2 "Background", as well as the involved technologies in Chapter 3 "State of the art". The Background Chapter also presents a general view of the state of machine learning in IoT explaining the main concerns of the current methodologies whereas the Chapter "State of the art" introduces the technologies shown in this thesis, how they are used currently and their role in this thesis.

Chapter 4 "Intelligence Layer Overview" describes the Intelligence Layer idea, providing an overview of it and the principal components of this thesis: the *Device Intelligence Layer*, *Intelligence Apps* and *Intelligence Services*. An improvement of the component definition and their interaction is addressed in Chapter 5 "Implementation". This chapter is related to the next one, Chapter 6 "Evaluation", which provides a first deep analysis of some weaknesses in the implementation, such as error handling. In addition, Chapter 5 presents an evaluation of the solution focusing on the main objective of the thesis: decoupling Intelligence from applications. The final results are analyzed in Chapter 7 "Discussion". Finally, the thesis ends with the Conclusions Chapter, where the key findings are explained and some recommendations are given for future research.

Chapter 2

Background

2.1 Intelligence

Intelligence can have an abstract meaning therefore this section will clarify the definition of Intelligence within the scope of this thesis.

Intelligence refers to a function that can be executed, they can be a simple algorithm or complex machine learning method. It is mainly a function to achieve one goal with clear input and output. For instance, a resize function, the input is an image and desired width and height, the output is an image with that size.

Therefore Intelligence is a function with a clear purpose, input and output. In section " Definition of Intelligence Service" and "Device Intelligence Layer, the Intelligence Apps and Intelligence Services Roles" the reader can find more explanations about how this thesis describes Intelligence. This thesis focuses on machine learning functions.

2.2 Sharing Intelligence across platforms

Currently, Intelligence is developed and published every day. This section focuses on the current method to make this Intelligence available but first, it is necessary to explain two key factors of how the Intelligence is represented: computational graph and weights. The graph is the flow that the input data has to go through, in other words, the calculus to perform over the data and the order, the weights are the values of these calculations.

The current state of the art to make the models available is to group the weights together in a public server, it is usually named "model zoo" [2], and develop the graph in frameworks, such as Tensorflow [3] or scikit-learn[4]. The main factor of this approach is that different developers can train and

share the weights. Nevertheless, it implies some limitations, such as changes in the framework and updating the Intelligence costs time and effort. The thesis investigates how to solve this having a deep look in the Intelligence Layer approach.

2.3 Current State of Machine Learning in IoT.

IoT is the idea of all connected world, it extends the internet connection among all the everyday devices such as smartphones, sensors, electrical household appliances, etc.

These devices would increase in number -about 15-30 billion [5]- and would produce a huge amount of data, it means that there is a machine learning niche in this area due to the requirement of the data to be either stored or analysed automatically.

Currently, most of these data have been processed by a cloud platform which is the predominant approach in IoT Intelligence. The main reasons for that are the need for a powerful computational effort and a big cost for developers to train and adapt models but this model is not feasible due to connectivity cost, data protection policies and the expensive changes during the time to maintain the system.

Generally, in the cloud-based vision, the server harvests the data, preprocesses it, generates machine learning models and takes care of deploying those models to the production environment, where devices use the models with local data. This approach implies that devices must contain machine learning code which has to be synchronized each time that a new model is deployed which means a cost in effort and time.

This thesis will investigate how to avoid the application synchronization with the cloud at deploying models. The research will focus on the Intelligence Layer which proposes one way to move the Intelligence from the application to one external layer, therefore the application just needs to call that layer to get the result and the application does not need any machine learning code either synchronization as it will be done by that layer. This layer will act as a service in the device, therefore, every application could call it.

Chapter 3

State of the art

3.1 Protocol buffers

Protocol buffers [6][7] are a method to serialize and deserialize data structures developed by Google. The Protocol Buffers design aims for performance and simplicity being smaller and faster than XML [8]. The proto files (.proto) contains messages which are the definition of the data structures. When proto files are compiled, it produces code to access to the information of the data structure, which can be used by applications to send or receive the data.

Note that all messages are serialized into a compact, forward-compatible and backwards-compatible but not self-describing binary format. it means that all the fields are not known without the external definition. This is an example of how protocol buffers works.

Figure 3.1: The protocol buffer example

In this thesis protocol buffers are used to represent machine learning models in ONNX, besides the writer sees a big potential to use them to create the information object used in Smart Functions.

3.2 Open Neural Network Exchange (ONNX)

ONNX [9] is an open-source format to represent deep learning models. ONNX provides a definition of an extensible computation graph model, as well as definitions of built-in operators and standard data types. Each computation dataflow graph is structured as a list of nodes that form an acyclic graph. Nodes have one or more inputs and one or more outputs.

Today, each framework has its own representation of the graph, though they all provide similar capabilities - meaning each framework is a siloed stack of API, graph, and runtime. Furthermore, frameworks are typically

optimized for some characteristic, such as fast training, supporting complicated network architectures, inference on mobile devices, etc. It's up to the developer to select a framework that is optimized for one of these characteristics. Additionally, these optimizations may be better suited for particular stages of development. This leads to significant delays between research and production due to the necessity of conversion.

By providing a common representation of the computation graph, ONNX helps developers choose the right framework for their task, allows authors to focus on innovative enhancements, and enables hardware vendors to streamline optimizations for their platforms.

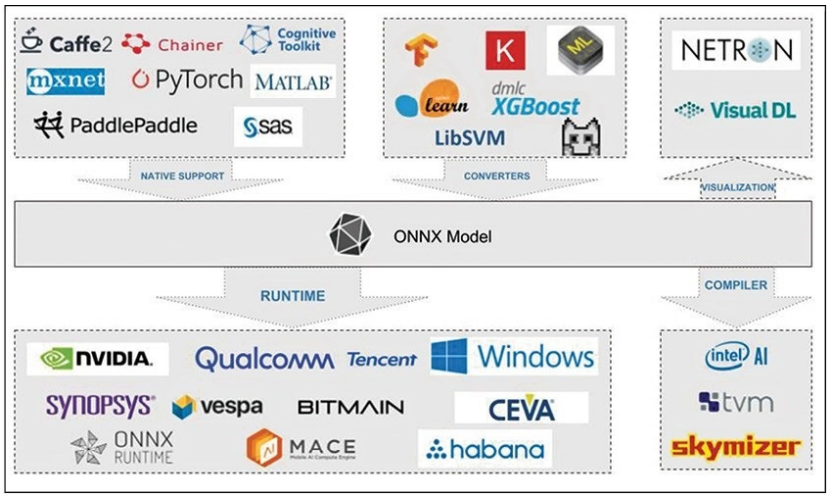

Figure 3.2: ONNX ecosystem

All the Onnx syntax is defined in proto files since Onnx files are generated by protocol buffers messages. Therefore, the generated files are binary encoded and non-human readable. It is necessary to serialize and deserialize the files in classes in order to read and create Onnx files, to perform this, the protocol buffers compiler must generate the classes from the ONNX proto files. This is an example of a node definition of the computational graph:

Listing 3.1: ONNX node definition

```
message NodeDef {
  // The name given to this operator. Used for naming inputs,
  // logging, visualization, etc. Unique within a single GraphDef.
  // Must match the regexp "[A-Za-z0-9.][A-Za-z0-9_./]*".
  string name = 1;
```

```
// The operation name. There may be custom parameters in attrs.
// Op names starting with an underscore are reserved for internal
    use.
string op = 2;

// Each input is "node:src_output" with "node" being a string
    name and
// "src_output" indicating which output tensor to use from
    "node". If
// "src_output" is 0 the ":0" suffix can be omitted. Regular
    inputs
// may optionally be followed by control inputs that have the
    format
// "^node".
repeated string input = 3;

// A (possibly partial) specification for the device on which this
// node should be placed.
// The expected syntax for this string is as follows:
//
// DEVICE_SPEC ::= PARTIAL_SPEC
//
// PARTIAL_SPEC ::= ("/" CONSTRAINT) *
// CONSTRAINT ::= ("job:" JOB_NAME)
//              | ("replica:" [1-9][0-9]*)
//              | ("task:" [1-9][0-9]*)
//              | ("device:" [A-Za-z]* ":" ([1-9][0-9]* | "*") )
//
// Valid values for this string include:
// * "/job:worker/replica:0/task:1/device:GPU:3"(full
    specification)
// * "/job:worker/device:GPU:3" (partial specification)
// * ""                                    (no specification)
//
// If the constraints do not resolve to a single device (or if
    this
// field is empty or not present), the runtime will attempt to
// choose a device automatically.
string device = 4;

// Operation-specific graph-construction-time configuration.
// Note that this should include all attrs defined in the
// corresponding OpDef, including those with a value matching
```

```
  // the default -- this allows the default to change and makes
  // NodeDefs easier to interpret on their own. However, if
  // an attr with a default is not specified in this list, the
  // default will be used.
  // The "names" (keys) must match the regexp "[a-z][a-z0-9_]+" (and
  // one of the names from the corresponding OpDef's attr field).
  // The values must have a type matching the corresponding OpDef
  // attr's type field.
  // TODO(josh11b): Add some examples here showing best practices.
  map<string, AttrValue> attr = 5;

};
```

3.2.0.1 ONNX Runtime

ONNX Runtime [10] is an ONNX model scoring engine focused on the performance. It offers a complete implementation of all ONNX operations, over the 1.2 version, within compatibilities.

ONNX Runtime is available for Linux, Windows, Mac with Python, C#, and C APIs. This thesis uses Android for developing and Java as a programming language, unfortunately, there is not an ONNX runtime Java API, thereforeit is necessary to use the Java Native Interface (JNI) to allow running C code.

3.3 Java and JNI

Java [11] is a popular language developed by Sun Microsystems that is now owned by Oracle. The Java key feature is the idea of "write once, run anywhere" (WORA), therefore the compiled code will run in any platform that supports Java, which makes the program code portable across platforms.

The Java code is converted to bytecode and any JVM can interpret that. Java is widely used for developing Android apps, in this context, the applications are executed in a special virtual machine: Dalkiv VM.

This thesis is not going deep into the Java features as it would be outside the scope of the thesis but it was needed to highlight the most important features related to this thesis: the developed Java code is platform-independent, therefore, can be reutilized for different approaches and the Java Native Interface.

In the previous section of this research, this thesis mention that the ONNX runtime has not a Java API, it would be a huge inconvenient if JNI

would not exist.

JNI [12]is a library developed with the purpose to include and execute libraries or application written in other programming languages, in this research a C library with the ONNX Runtime was incorporated into the project. This is also portable as it will run in other JVM.

3.4 Inter-process communication

This thesis proposes an implementation of the Intelligence Layer 4.1 involving different actors that communicate with each other in the same device through Inter-process communication (IPC).

IPC [13] is the mechanism that different components use to share data. There are two main methods to perform IPC: **shared memory** and **message passing** [14][15].

Analyzing both approaches [16] and due to the IoT context of this thesis, **message passing** suits better as some IoT devices memory resources are low and the thesis aims to encompass all of them.

In a message-passing method, the actors send messages following a specific protocol over a communication link that needs to be established. This thesis research about how the exchanging of messages should be done and give some insights about how it can be developed in Android, having in mind the requirements 1.4.

3.5 Android

Android [17] is a well-known open-source operating system made by Google. The OS is based on Linux and designed for touchscreen devices, such as smartphones and tablets although nowadays there are different versions for other platforms in different areas, for example, TVs, smartwatches and the automotive sector.

3.5.1 Android Architecture

Android has is a multi-layer architecture based on Linux Kernel with an Android runtime on top of it and an application framework running on top of the runtime. The components of each layer and the layer itself are personalized to achieve the optimal performance for the environment. A short explanation of each layer is provided here. A visual representation of the architecture is shown in 3.3.

Figure 3.3: Android Architecture

- Linux Kernel. It is the base of the architecture, there are the hardware drivers to manage the devices as well as processes, networking and memory management, in conclusion, it is a layer to manage the device hardware.
- Android Runtime (ART) and Android Libraries ART is the Runtime environment to compile the apps. To do this task the apps are transformed in bytecode to machine code by AOT (ahead-of-time) at installing time. One important component of this runtime is the Dalvik Virtual Machine which acts similar to the Java Virtual Machine (JVM) and is designed especially for Android. Besides the runtime, there are a large set of Android libraries based on Java for Android development. They are mostly focused on visual tasks such as graphics drawing.
- Application Framework. The application framework provides a set of high-level libraries that aim to run and manage all the applications. In Android all the applications should be replaceable, reusable and interchangeable components, this framework takes care of this purpose. All developers should use the components of this framework to run their applications.
- Applications The applications developed should be run and installed here to use the powerful previous layers. It includes all the applications, native and not natives.

3.5.2 Choosing between a service and a thread

The main difference between services and threads is the user interaction with the application. Services should be chosen when there is no user interaction. If the user interacts with the apps, threads are the preferable choice. Note that services will run in the same thread as the application so the service should create a thread to prevent blocking operations.

3.5.3 Services

1. **Overview**

   A service [18] is an application component that can execute long-running operations in the background without user interface. Once the service is started by an application component it continues running in the background and it does not matter if the user runs another application meanwhile. Besides, there can be a component among the service and applications, this component binds the service and can be used to interact with it. In Android there are three kinds of services:

   - Foreground:
     Foreground services continue running although the app is in idle, it performs some task which has to be displayed some notification
   - Background
     Contrarily to the foreground service, the background service does not need to inform the user. The service will be still running until it is destroyed itself.
   - Bound
     This service has a client-service interface, therefore the components can interact with the service using this interface. To bind a service is it needed to use the bindService() function. This service will run just when application components are bounding it.

2. **Life Cycle of a Service**

   To start a service use startService(), it will a call to the service's onStartCommand() method. Once the service start, it is independent of the component that had called it therefore if that component is destroyed the service would run. To send data to the service is it used Intents with the necessary data, this intent is the same as the one in onStartCommand() To stop a service, any components can call stopService() besides the service can stop itself with stopSelf()

3. **Background Execution Limits**

   It is a new feature in Android from version 26 "Oreo". As it is explained in previous sections Android is developed for battery-powered devices and it focuses on keeping the battery consumption a minimum. This feature is related to this, usually, a lot of background services are running using much power and memory resources affecting the user experience. Therefore from the Android API 26 version, background execution limits were imposed. Apps are restricted in two ways:

   - Background Service Limitations: There are limits to using background services while an app is idle. Foreground and bound services are not affected by this. More specific, apps in the background have a small window to create and execute services while apps which are in the foreground can call and execute both background and foreground services without limitations. the app is in idle at the end of the window and it is stopped and destroyed in the same way if the service calls the stopSelf() method.

     In other words, background apps are not allowed to create a background service. Therefore, "Oreo version" included startForegroundService() to start a new service in the foreground. it creates the service and it has the small window (5 seconds) to call startForeground() and showing a notification to the user, if not it is stopped and the app is ANR.

     Android identifies between foreground and background apps. To consider an app to be in the foreground has to accomplish any of the following:

     – It has a visible activity, whether the activity is started or paused.
     – If the app has a foreground service.
     – Another foreground app is connected to the app or has a bound component.

     Therefore a background app does not accomplish any of them.

   - Broadcast Limitations: This change was made since apps which are listening system broadcast events consume a lot of resources as all are triggered. Therefore from "Oreo" version, apps must register for implicit broadcast at runtime and, apps cannot use their manifest. -implicit broadcasts do not target a specific app so it is sent to all the system, on the contrary, the explicit broadcast target a specific app-. This limitation does not apply for broadcast with signature permissions as the messages are sent just to

the apps which have signed the certificate There are two ways to register a receiver:

- At runtime, apps should call registerReceiver() for any broadcast, whether implicit or explicit.
- In the manifest, just explicit broadcast can register there.

4. **Implement a Service in Android**

   Android offers different implementations of services: Services, Intent Service and JobIntentService. To declare that your application has a service you should add it in the manifest file.

Listing 3.2: Android Manifest example

```
<manifest ... >
  ...
  <application ... >
      <service android:description="string resource"
         android:directBootAware=["true" | "false"]
         android:enabled=["true" | "false"]
         android:exported=["true" | "false"]
         android:icon="drawable resource"
         android:isolatedProcess=["true" | "false"]
         android:label="string resource"
         android:name="string"
         android:permission="string"
         android:process="string" >
    . . .
</service>
      ...
  </application>
</manifest>
```

There are some attributes which set the behaviour of the service, among all of them the most related to this thesis are:

- **directBootAware**: it allows the service to run before the user unlock the device
- **enabled**: it sets if the system can instantiate the service, default is true.
- **exported**: if true the service can be called by other applications and "false" if just the same application component can invoke it.

- **isolatedProcess**: Isolate the service from the rest of the system (when it is "true")or not (set to "false"), Binding the service is the only way to communicate with it.

Service, It is the custom service in which the developer has to be aware of all the live cycle of the service and the background execution limits, it is a huge deal but this way offers the flexibility to personalize the service to adjust your requirements such multitasking and start-stop. To implement a Service, it has to extend the Service class:

Listing 3.3: Normal Service example

```
public class HelloService extends Service {

@Override
  public int onStartCommand(Intent intent, int flags, int
      startId) {
      Toast.makeText(this, "service starting",
          Toast.LENGTH_SHORT).show();


      return START_STICKY;
  }
```

As it was explained it uses onStartCommand() therefore this method should be filled with the desired operations.

IntentService is more abstract than the normal Service, it managed all the intents in a queue and it executes one single thread to perform them, therefore onHandleIntent() should be implemented instead of onStartCommand(). This service will stop once that the queue is empty so it is not necessary to call stopSelf()

Listing 3.4: IntentService example

```
public class HelloIntentService extends IntentService {

  public HelloIntentService() {
      super("HelloIntentService");
  }

  /**
   * The IntentService calls this method from the default
       worker thread with
```

```
     * the intent that started the service. When this method
        returns, IntentService
     * stops the service, as appropriate.
     */
    @Override
    protected void onHandleIntent(Intent intent) {
        // Normally we would do some work here, like download a
            file.
        // For our sample, we just sleep for 5 seconds.
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            // Restore interrupt status.
            Thread.currentThread().interrupt();
        }
    }
}
```

JobIntentService, when the background execution limitations were introduced in the Oreo version, they made more difficult to work with IntentService, therefore, this class were implemented to solve that problem. It basically works in the same way that IntenService but uses a JobScheduleAPI to run background tasks. The jobs loaded through JobScheduler are executed sequentially and JobIntentService would manage Wakelock.

Listing 3.5: JobIntentService example

```
public class MyJobIntentService extends JobIntentService {
    @Override
    protected void onHandleWork(@NonNull Intent intent) {
 // Normally we would do some work here, like download a file.
      // For our sample, we just sleep for 5 seconds.
      try {
          Thread.sleep(5000);
      } catch (InterruptedException e) {
          // Restore interrupt status.
          Thread.currentThread().interrupt();
      }
    }
public static void enqueueWork(Context context, Intent
    intent) {
        enqueueWork(context, MyJobIntentService.class, JOB_ID,
```

```
            intent);
    }
}
```

3.5.4 Conclusions of Android

Analyzing Android under the scope of the thesis objectives, Android offers most of the features needed for developing such a new approach. To accomplish the objectives, it is needed a platform able to be personalized to our requirements and Android fits so well due to its open nature approach, besides the big Android community would be interested in this research and support it. Android has also some withdraws due to the background execution limitation imposed from the "Oreo" version, this thesis will also study the way to adjust the idea of the Intelligence Layer with these limitations.

3.6 Analysis of the state of the art

At this stage, all the key technologies involved in the thesis are explained. They have shown us that it is possible to inference Intelligence easily using ONNX and there is a suitable OS to inference them on IoT devices but the only problem is to accomplish this job more efficient due that currently, all this Intelligence belongs to the application and it implies that the application should be adjusted due to the Intelligence requirements.

The Intelligence Layer proposes a solution for this, abstracting the deployment of the Intelligence in one layer which takes care of the Intelligence orchestration.

In the next section, the Intelligence layer proposal is presented, together with a feasibility assessment using current technologies and the exploration of new approaches.

Chapter 4

Intelligence Layer Overview

4.1 Intelligence Layer

As explained in section 2, currently with the cloud-based approach, these applications which use machine learning, that Intelligence is integrated into the application, therefore the Intelligence belongs to the application. There are some issues with this vision, the main problem is that all of the scopes are defined, it means that it would just under these scopes that cannot be changed but in real life, the needs can change and some variations have to be made, it implies a lot of effort in time and resources.

In order to solve that problem, the Intelligence Layer proposes that Intelligence should be separated from the application. It should be orchestrated by one external layer that provides Intelligence for all the applications in the device. Therefore, all the actors of the system change roles and now the device becomes a loader of Intelligence in applications by the Intelligence Layer Orchestrator.

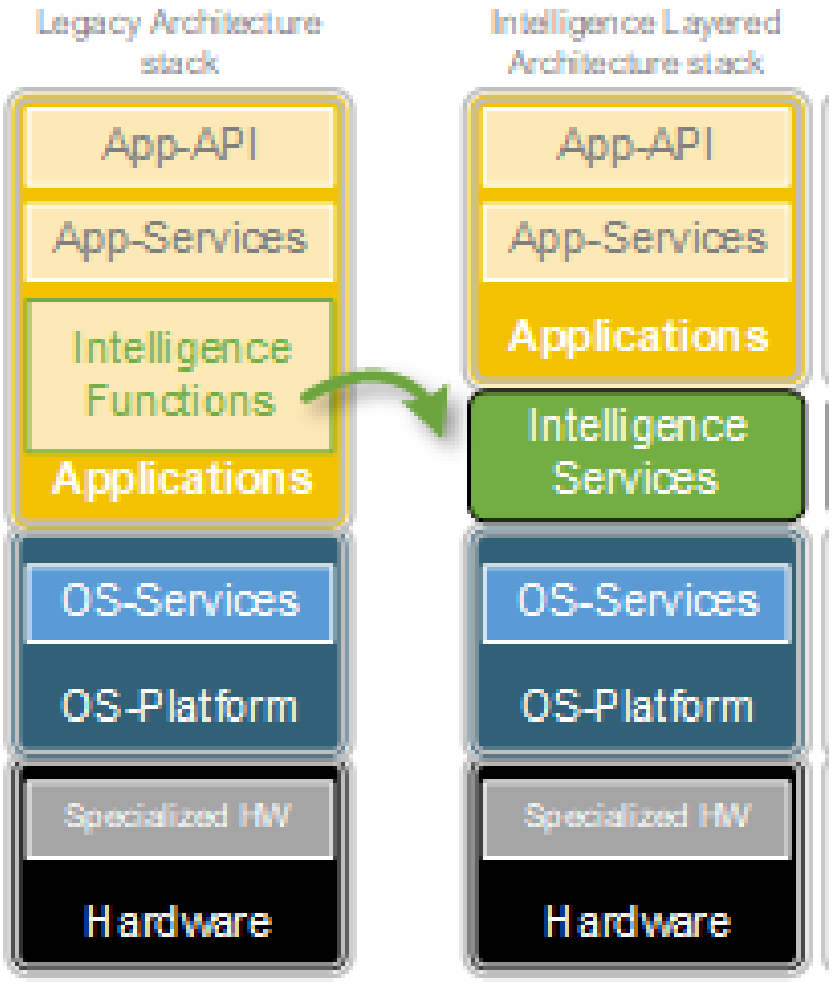

Figure 4.1: Moving Intelligence from apps

4.1.1 Summary of the Ecosystem and Actors

This Intelligence Layer also implies more external actors over the system, as it is shown for the whole ecosystem in Figure 4.2.

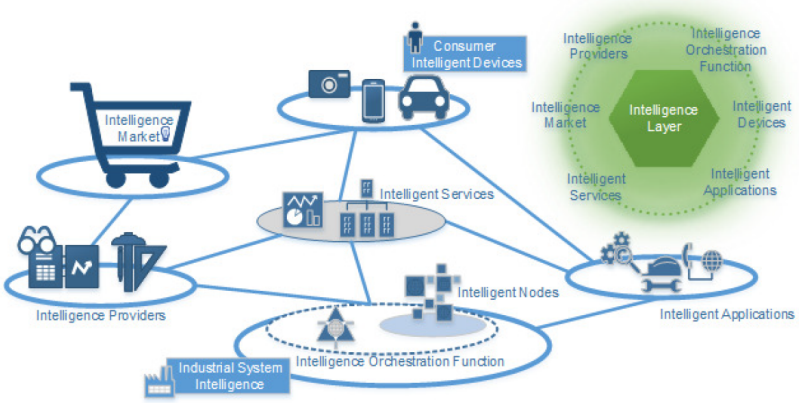

Figure 4.2: Intelligence Layer Overview

Figure 4.2 shows how big this proposal is. The research of all the actors in the system will go out of the thesis scope. Therefore, the Consumer Intelligence Devices, Intelligent Applications, Intelligence Orchestration Function and Intelligent Services nodes are mainly targeted in this thesis. Anyway, a brief description of all the actors would be given to the reader to provide the best understanding of the Intelligence Layer.

- **Intelligent Applications** They are located at the application layer of the devices, they would request the Intelligence from the Intelligence Layer and would get the Intelligence result.
- **Intelligent Services** They execute the Intelligence for the applications. The applications should be registered in the to use then and it should interact with the Intelligence Marker. Although this service is personalized due of the different approaches just Intelligence Services running in the same device as the requesting applications are under the scope of this research.
- **Intelligence Orchestration Function** This actor will orchestrate the Intelligence for applications. it takes care to retrieve the Intelligence services from the market, make them available for the Intelligent Applications and inference. Besides, it is aware of the system architecture therefore it ensures that all of the previous tasks are performed due to the architecture requirements. In this thesis, the writer names it as Device Intelligence Layer (DIL).
- **Consumer Intelligence Devices** They include all the devices able to perform Intelligence features. In this thesis, just Android devices are

included and the device contains the Intelligence Orchestration Function, Intelligent Services and Intelligent Applications in the way that the Intelligence is provisioned and executed due to the application requirements.

- **Intelligence Market** It provisions Intelligence. It is in charge of the delivery of Intelligence from the network. It should inform the DIL to new updates of the IS and make available external resources, such as external data or new IS.

  This market should categorize the Intelligence due to its scope (platform/target/ etc...).

- **Intelligence Providers** All the developers willing to generate Intelligence are in this node. They develop and improve Intelligence. At provisioning the Intelligence to the Market they are required to include all the information and steps needed to inference it.

Analysing Figure 4.2 under the scope of the thesis, the actors can be classified in external actors and internal. In the Figure 4.2 system the actors executing inside the device are internal as they act inside the same platform and the external actors are the remainder because they do not belong to the device itself and they belong to other platforms.

The internal actors are Consumer Intelligence Devices, Intelligent Applications, Intelligence Orchestration Function and Intelligent Services. External actors are Intelligence Market and Intelligence Providers.

As the thesis scope is on devices area, the research is focused on the internal actors and for clarifying purpose inside them it can be distinguished between Device Intelligence Layer and Intelligence Applications. The Device Intelligence Layer would act as the Intelligence orchestrator for the Intelligence Applications in the way that they can request for Intelligence and the Device Intelligence Layer would be in charge of requesting, inferencing the Intelligence and posting the inference results. Therefore, inside the Consumer Intelligence Devices are running the Device Intelligence Layer, which is composed of the Intelligent Applications, Intelligence Orchestration Function and Intelligent Services, and the Intelligence Applications.

Note that this paradigm would be different if the internal actors become external ones using a web service approach but this is not the goal of the thesis.

4.1.2 Device Intelligence Layer, the Intelligence Apps and Intelligence Services Roles

Defining the roles of the actors is important to continue this research, in rough outlines, the Apps would act as a client, the DIL as a resource enable service and the Intelligence Service as the resource.

- Apps: they request the Intelligence to the DIL. This approach proposes that the Intelligence does not belong to Apps and they just need to manage the communication with the DIL in order to send requests and get the responses. To perform the inference of the Intelligence Services, the Applications have to register in the DIL getting the authorization token.

  The applications have to be developed in a proper way to communicate with the DIL and utilize the IS results. The thesis will research a proper communication method among the DIL and the Application and an approach to assist them to utilize the output of the Intelligence services.

- Intelligence Services: they are the resource the DIL provides to Intelligence Applications. They describe how the Intelligence should be executed (API), the required input, the output, the purpose and some requirements, for instance, the runtime. The Intelligence Services are composed of one or more Atomic Functions.

  The Atomic Functions are no divisible functions which perform one task for the specific input to get a specific output. They are represented by an Information Object which contains all the information required to inference the Atomic Function.

  It seems that Atomic Functions and Intelligence Services are the same and actually they are, basically an Intelligence Service is a composition of different Atomic Functions, see this is example:

  There are three Atomic Functions:

Atomic Function	**Input**	**Goal**	**Output**
Resize_Function	Image*, size.	Resize	Image with size
Object_Recognition	Image with size	Object Recognition	Class Probabilities
Get_Labels_Function	Probabilities, classes	Get Most Likely Class	Class*

  The Intelligence Service which represent this three Atomic Function is the following:

	Input	**Goal**	**Output**
Object_Recognition_Intelligence	Image*	Object_Recognition	Class*

How to make the orchestration of Atomic Functions to generate the proper Intelligence Service is a topic of another Thesis, the scope of this thesis assumes that the Intelligence Services are already defined properly. This thesis will research about how to use and define the API of the Intelligence Services.

- DIL: this layer is an orchestrator service in which Intelligence Services can be plugged/unplugged due to the registered app requests. This service has to be able to:
  - Discovery of Intelligence. The DIL should be able to request the IS to fit the App requirements. This thesis would assume that the Intelligence is already in the device therefore it is not necessary to query for them in the Intelligence Market.
  - Check the policies of the IS with the device and user policies at querying to ensure matching results.
  - Register Apps in the DIL. Once that the DIL has a proper IS for the App, it should register the "IS-App" relationship to allow applications to recall the Intelligence Service.
  - Handle recalls: Once that the application is registered, the DIL has to be able to do the inference of the IS recalling the "IS-App" association and sending back the results.

  Therefore, the DIL should be an orchestrator which serves Intelligence and communicate with the apps. In other words, it would be a service in which the applications can register and request the Intelligence Services making "App-Intelligence" associations to make them available for the application at recalling. The Applications can request more than one IS and one IS can be used by different Applications, the DIL has to manage all of this process.

  Note that the user of the app can trigger part of this process and the user also should take some decisions, for example, he could choose the Intelligence to use and change the policies to allow the DIL to use different Intelligence Services.

There are some unknowns in this process, such as "how is the communication channel?", "what is the protocol?", "how is it exposed?". In the next section, these questions are studied more in detail.

Chapter 5

Implementation

5.1 Definition of Intelligence Service

The ISs are the key resource in this thesis, without them the idea of the Intelligence Layer would not be possible. As it was explained in the previous section 4, they are defined as a function able to achieve a purpose with defined input and output. Although this is not enough for discovering as it can reach ambiguous results, this section focus on the improvement of this definition using semantics.

The main goal is to improve this description providing to the Intelligence Services a definition which focuses on the meaning and have to be unambiguous. For example, the definition of a cat is the same in the globe although it is named in different ways. The proposed solution for that is to use the Semantic Web to keep the real meaning of the Intelligence Service components: Input, Output and Goal.

The principal advantages of applying semantic [19] is that the data becomes self-explanatory making it unambiguous [20] and more easily discoverable. In addition, the semantic database uses a graphing approach which simplifies including changes and information or metadata. To represent semantic data there are two main technologies: Resource Description Framework (RDF)[21] and Web Ontology Language (OWL) [22]. The description of the IS are represented in an RDF schema as it is more flexible than OWL as RDF [23] just describes the data structure and their relationships whereas the OWL describes the semantic relationships.

RDF is just one way to represent the IS, in the context of the Intelligence Market, RDF is used as a database that contains the data model of the IS and SPARQL [24] as the language query. This method allows performing queries without knowing the data model, therefore the query would be performed

taking into consideration the main definitions: Input, Output and Goal. As the goal of the thesis is the DIL communication and the IS inference, JSON is used to represent ISs even though it presents unstructured issues. In the Intelligence Layer ecosystem, all the actors are inside a reliability context, therefore all the actors are aware of the proper JSON format and know how to process it. In addition, there are already many services working with JSON-based communication and less reliability context.

Outside this thesis, the search engine is provided by the Intelligence Market. Both of the representations are compatible witch each other as there is a W3C syntax for the conversion [25]. Although the W3C community recommends using JSON-LD [26], which is a JSON with data interoperability features for Web-scale, this thesis would still use JSON as the semantic definitions are not properly defined yet and the extension from JSON to JSON-LD would be pretty easy in the future.

In addition, as the desired platform is Android, JSON is the preferable way and there are no proper tools to handle semantics in this platform [27].

Therefore the IS Definition can be divided in:

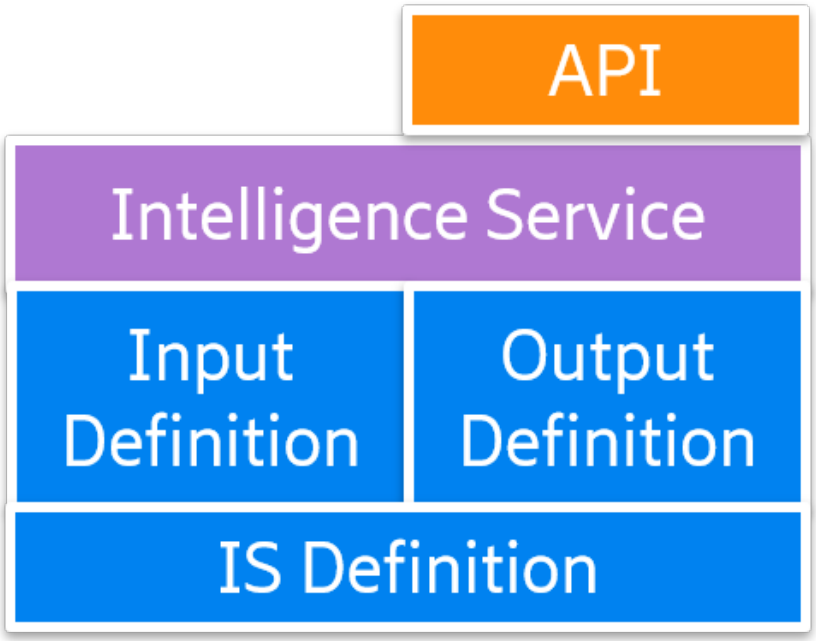

Figure 5.1: Intelligence Service

- Intelligence Definition

  - Intelligence task/goal: what is required from the Intelligence to achieve, for instance, object recognition.
  - Domain: the context in which the Intelligence service would be executed, for example, Automotive Traffic.

- Input Definition
  - Input data: data to be used in the process, for example, images, raw data, speech...
  - Input Denotation: metadata of the input, for example in the case of pictures this field can be RGB or BGR

- Output Definition
  - Output Required: output expected from the Intelligence, for example, it can be the location of the objects in a picture.
  - Output Denotation: more information about the format of the output, for example, the developer wants to get the labels in different languages.

- API: It is the key to inference the Intelligence Service. It contains the pipeline of Atomic Functions, the requirements and the metadata of the IS. The DIL has to interpret this in order to fulfil dependencies and perform the inference in the IS.

Listing 5.1: IS API description

```
Info={
    "id": unique IS id,
    "name": name of the IS,
    "hashSignature": signed IS by a reliable Intelligence provider
       or not, boolean value
    "nativSup": the device support it or not, boolean value
    "externalResources":[
       "value": it is needed or not, boolean value
       "resource": link to access the resource
       ]
    "scopeApplication": context of the application: speech, images,
       raw data, etc
    "modelPath": path of the machine learning model, ONNX in the
       thesis
    "transformations"=[
```

```
        "inputTransformations":[preprocessing of the IS, list of
            (atomic) functions to process the input data ],
        "outputTransfromations":[postprocessing the output of the
            machine learning model, list of (atomic) functions.]
        ]
}
InputInfo={
    "inputType": data type of the input such as image, raw data,
        music, etc
    "numInputs": number of inputs
    "inputSource": path to the data source
}
outputDefinitions=[ list of outputs
    {
    "dataStructure": structure of the data
    "dataType": type of data
    "dimensions": dimensions of the data
        "metadata":[
            {   "outputLabels":labels of the output
            }
        ]
    }
]
jobInfo={
"goal":object recognition, classification
}
```

Listing 5.2: IS API EXAMPLE - MNIST

```
Info={
    "id":001,
    "name":mnist,
    "hashSignature":0,
    "nativSup":1,
    "externalResources":[
    "value": 0,
    "resource": ""
    ]
    "scopeApplication": image
    "modelPath":/storage/mnist-model.onnx,
    "transformations"=[
        "inputTransformations":[[size:{dim:1,dim:28,dim:28}]],
        "outputTransfromations":[[softmax]]
```

```
        ]
}
InputInfo={
    "inputType":[image],
    "numInputs":1,
    "inputSource":[/storage/image.jpg]
}
outputDefinitions=[
    {
    "dataStructure":Array,
    dataType:float,
    "dimensions":[1,10],
    "metadata":[ {
        "outputLabels":[0,1,2,3,4,5,6,7,8,9]
        }]
    }
]
jobInfo={
"goal":object recognition, classification

}
```

5.2 Use cases of the DIL

The DIL should manage the IS to make them available on account of application requests. The applications requirements in an IoT device are highly correlated to the device purpose, for example, the there devices without user interaction, such as GPU mining for bitcoins, on the contrary, an Android device aims to be user friendly. In order to fit these requirements the DIL should be flexible enough to expose some functionalities and keep others in the background, see the following figure:

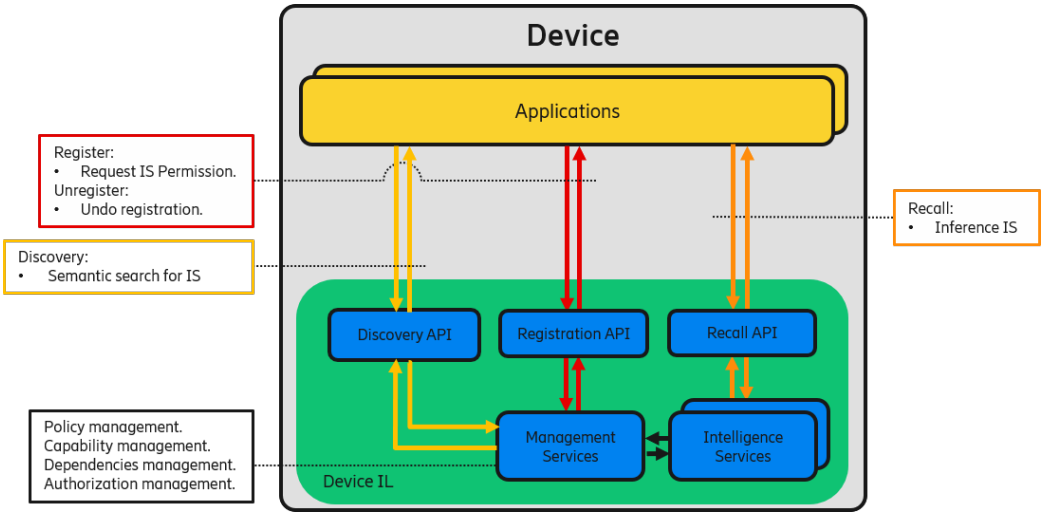

Figure 5.2: DIL Functionalities Architecture

As it is shown, the exposed functionalities are Discovery, Registration and Recall, they are the main functionalities to enable the DIL to orchestrate the Intelligence, nevertheless, in some contexts, the exposed APIs to the applications can change depending on the purpose of the device. The following architecture shows the use case when the user interaction of the device is low and the Discovery is implicit in the Registration but it remains in the DIL.

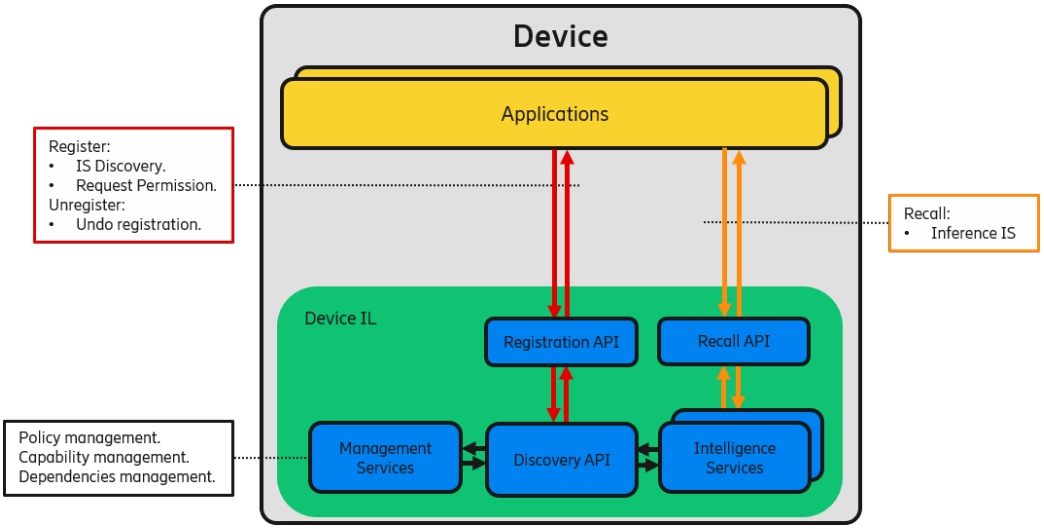

Figure 5.3: DIL Functionalities Architecture 2

They are the two main architectures for the DIL, in the first one the Discovery is explicit and in the second one, it is implicit.

- **Management Services**
  - Policy management. The devices and the applications could have its own policies, for example, the device uses confidential data, therefore, all the processing have to be performed in the device or the application cannot use more than a small percentage of the CPU. The DIL should act based on the polices to ensure the desired workflow of the system. These policies will affect the discovery and registration functionalities as one IS with incompatible policies could not be installed on the device.
  - Capability management. This service is aware of the sensors in the device and their status. It will provide the information related to the drivers and which data the sensor is able to collect. For example, if one Intelligence needs a colourful image the sensors installed on the device have to be able to collect that data. Besides, this module would check the sensor status to inform the DIL if they are ready to use or not.
  - Dependencies management. In order to perform the ISs, it is needed some software, this module ensures that all the requirements are ready o use for the DIL, for instance, it is needed a new runtime for ISs.
  - Authorization management. This module is in charge to generate the authorization for the application to use the Intelligence Services. Outh2

- **Discovery** The discovery allows the application to know the IS available. Semantic is used in this process, as it was explained in the anterior section the IS would be defined semantically to prevent the result to be ambiguous.

  Using this module the App can obtain IS definitions, querying for the desired Input, Output and Goal previously explained.

  This functionality is mainly provided by the Intelligence Market, therefore is it out of the scope of this thesis.

- **Registration** The registration case allows the DIL to manage the Apps and Intelligence Services in the device. The goal of the registration phase is to allow Apps to use the Intelligence Services, therefore in this phase, the DIL has to ensure that the device fulfils the policies and dependencies.

Once that the DIL is ready, it should create the APP-IS association to allow Apps to Recall.

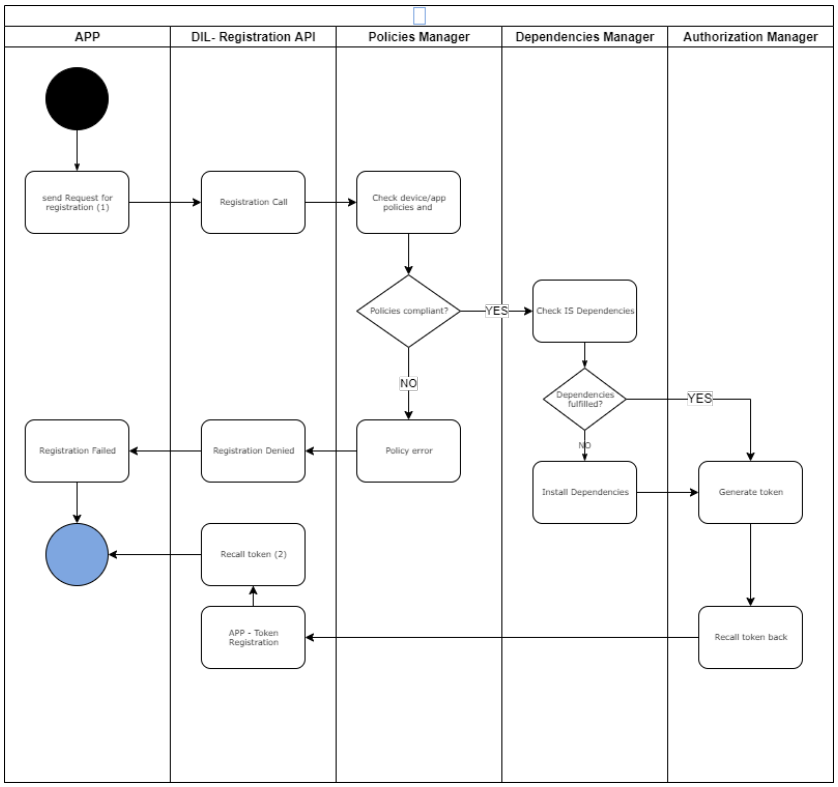

Figure 5.4: Use case - Registration

- (1) Request Data:
  - IS Description or unique ID.
- (2) Response Data:
  - Intelligence Profile:
    - Unique recall token.
    - Unique app ID.
    - IS API.

- **Recall** This is the final step of the process, once that the applications are registered in the DIL they are ready to recall the IS.

  This module knows how to use the IS API to perform the inference and sending back the results. Besides, it should be aware of the registration parameters to act based on them. For instance, if the application is

register to use one type of input definition, it has to trigger an error if the input does not fit the definition.

This call has to be customized therefore a communication paradigm is studied in the following section.

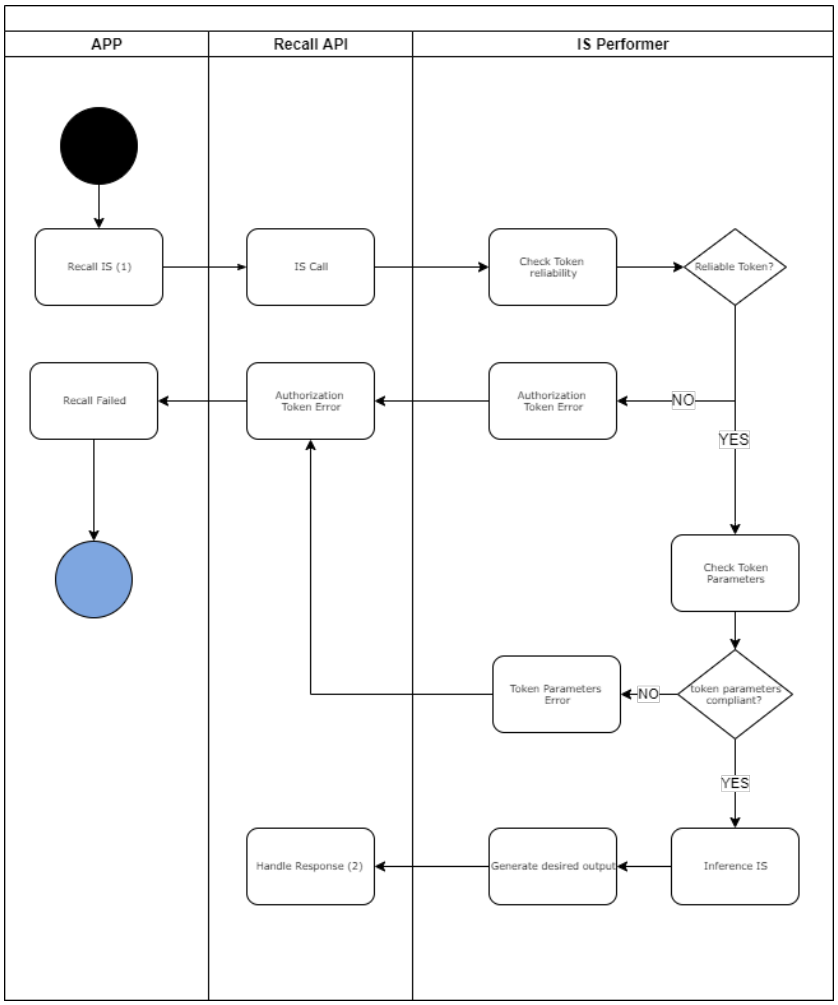

Figure 5.5: Use case - Recall

- (1) Request Data:
  - Unique recall token.
  - Unique app ID.
  - Input source.
- (2) Response Data:

* IS output. How to handle this response is studied in this thesis.

5.3 Communication among the DIL and the Apps

Communication is the key feature of every IoT system. Without it, all devices would become lonely working nodes, which would be the opposite of the "all connected world" IoT idea. In our case, the DIL has two communication areas: inside the device and outside the device, each one different actor take part in it.

This thesis will not research about the communication with the exterior, the participants on this interaction are external actors involving communication over the internet.

On the contrary, this section focuses on the IPC protocol among the DIL and the Intelligence Apps, studying the registration and recall cases within the data involved in the communication.

The context where the DIL can perform is so variety, therefore this study purposes a flexible IPC to allow the DIL to adapt in different scenarios.

5.3.1 Intercommunication Paradigm

This section studies the best way of interaction. In the context of the DIL, there are different actors sharing information, mainly the applications share information with the DIL but they do not do with other applications, therefore this communication channel should be point-to-point [28] between an application and the DIL. In other words, when an application sends information, only the DIL should receive it and in the case that the DIL send any information, it has to be sent to selected applications. Besides the DIL always has to send a response to the requester.

The most used ways to share the information are sharing memory block and message passing. The IPC protocol proposed in this thesis is based on a message-passing idea[15], in particular, it is a Request-reply pattern [29] with point-to-point channel [28].

The sharing memory is mainly used to coordinate different processes being difficult to escalate. In the DIL context, the protocol to manage the communication by sharing memory would be difficult as the apps could not change the memory anytime. In addition, the IoT context means that the memory availability maybe not high and the DIL would require connexion

with the exterior, therefore a messaging passing protocol fit much better in the context of IoT and the DIL. See the following figure:

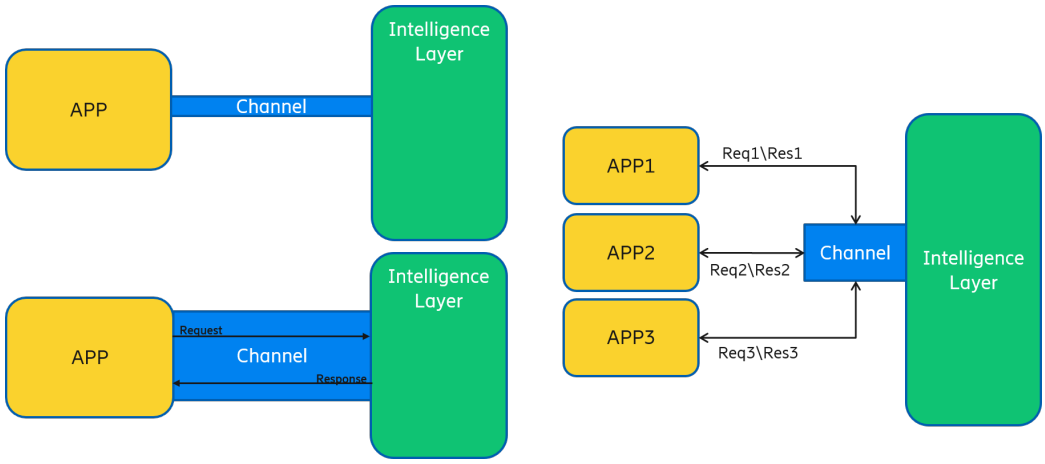

Figure 5.6: Architecture - APP-DIL channels

The messages in this channel have to have a specific receiver and the DIL has to be accessible by all the applications on the device. In the following sections the author studies about the IPC protocol and how the DIL should handle the messages.

5.3.2 DIL - Architecture to Request Handling

Due to the IoT context of the Intelligence Layer there are a large number of different case cases. The proposal aims to be flexible enough to handle messages depending on the app requirements. This section focus on how to handle requests in the DIL within the following elements:

1. The **thread manager** creates threads to perform the request. It should be configured to handle a maximum of threads depending on the device resources. It continuously checks the queue of request to handle the request or not depending on the number of current threads performing.

2. **Queue of request**, it queues all the request arriving at the DIL. The protocol to obtain the requests should be first in first out (FIFO) with some exceptions. MAX size?

3. **Response list**, The DIL would send the response to the Apps directly or not depending on the App needs, the DIL should save the Responses temporally.

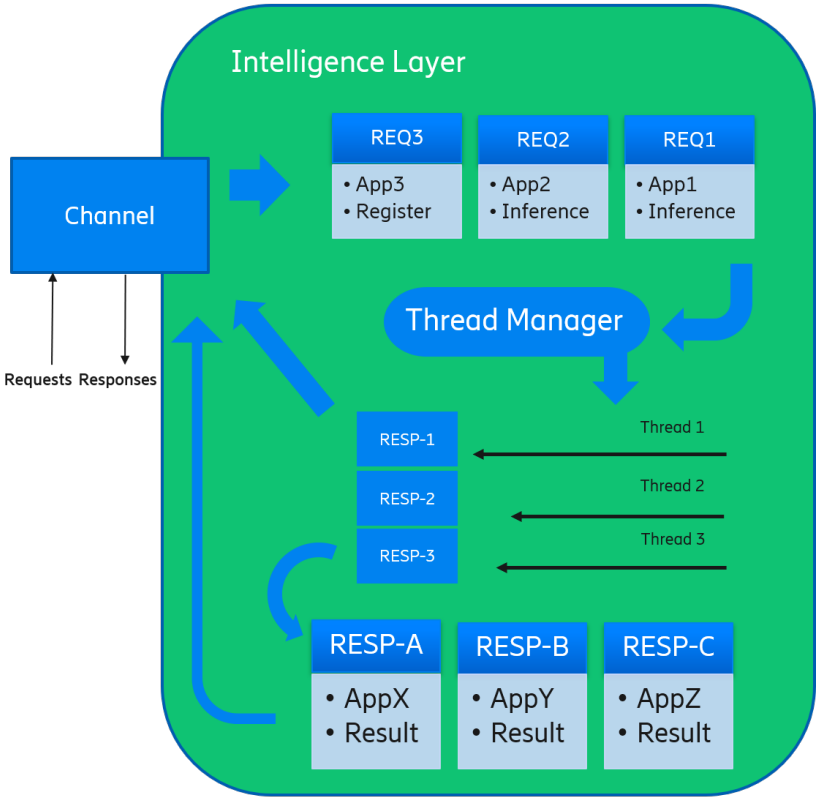

Figure 5.7: Architecture - DIL

Therefore, the workflow should be the following, once that a request arrives at the DIL, it should add the request to the request queue and the thread manager should constantly check the queue in order to create a new thread to handle the response or not depending on the number of current threads. When the thread finishes performing the request, the DIL will follow the protocols explained in the following section to send the result to the application.

5.3.3 IPC Protocols

This section purposes different protocols to send the responses back to the applications. Therefore when an app register in the DIL should select the best protocol to receive the responses.

Applications can utilize the DIL for many scenarios and maybe they are not always ready to handle a DIL response. Therefore the best way to act is to configure how the applications want to receive the responses and analyze if they can process the message or not is the key feature, accordingly, the purposed protocols are based on both scenarios.

1. **App is always ready to get the result (AR).**

    - **Synchronous behaviour (ARS):** app wait for response, therefore it is a blocking behaviour in the app side, therefor, this requests should be prioritized in the request call to avoid a high blocking time.

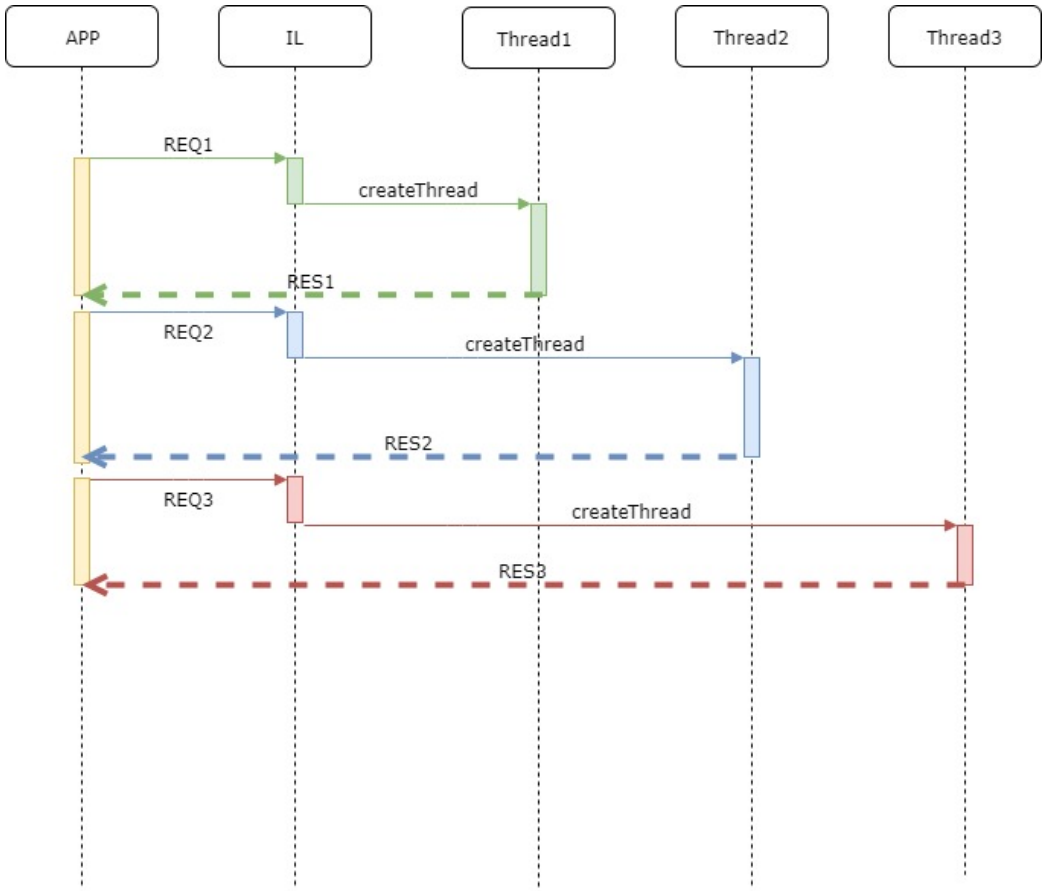

Figure 5.8: Sequence - ARS

- **Asynchronous behaviour (ARA):** app does not wait for a response. It implies that the app can send multiple requests and

the responses can send back in a different order as the request was sent. To solve this the apps should configure how the DIL should act:

– **Send responses in order (ARAO).** This case does not matter the order that the DIL finish performing the request that the App will receive than in the same order that it has sent them. To do so the responses would be stacked in the responses list until proper delivery time.

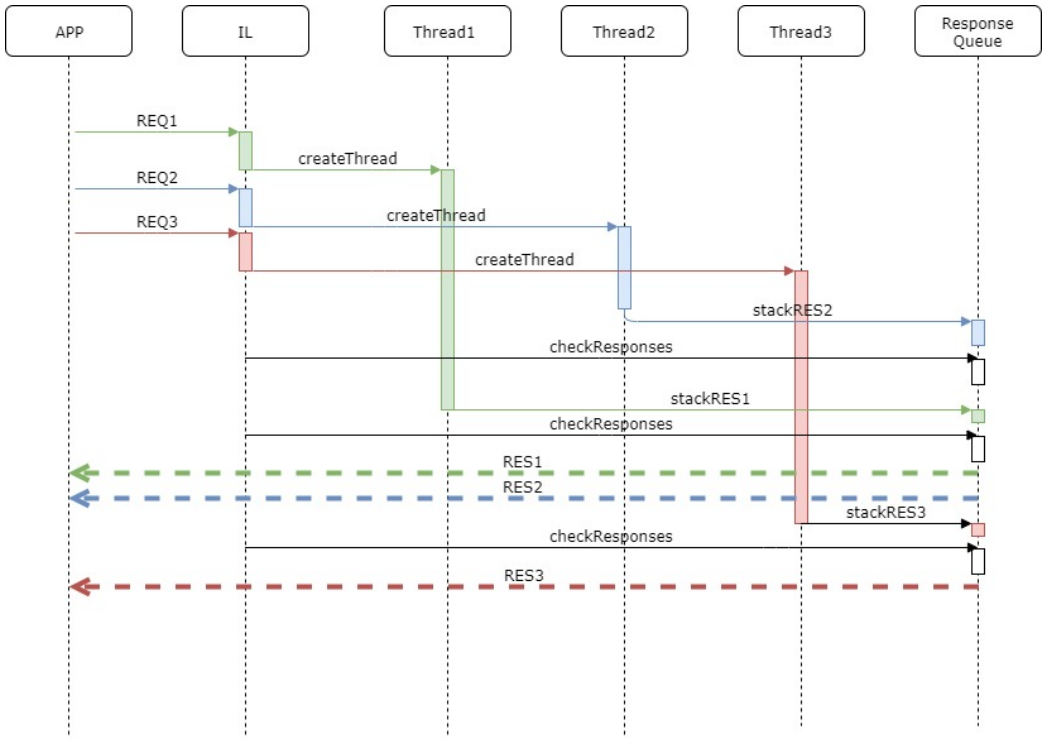

Figure 5.9: Sequence - ARAO

– **Execute the requests no mattering the order (ARANO).** This is the simplest case where the DIL will send the responses as soon as the thread ends.

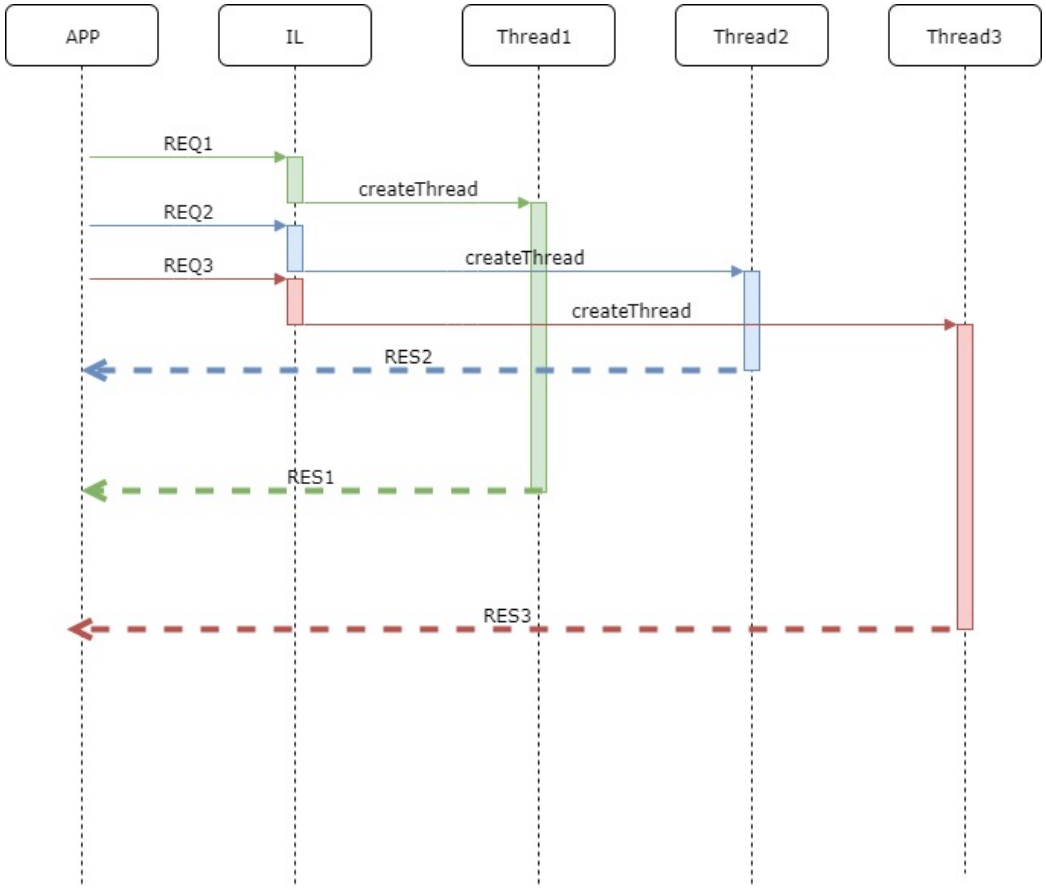

Figure 5.10: Sequence - ARANO

– **Process the last request only (ARAL).** To perform so, the DIL should only perform one thread of the same application and including at maximum one request in the queue, if there is already a request it will be replaced for the new one, therefore the DIL can have one thread and one request in the queue for the same application.

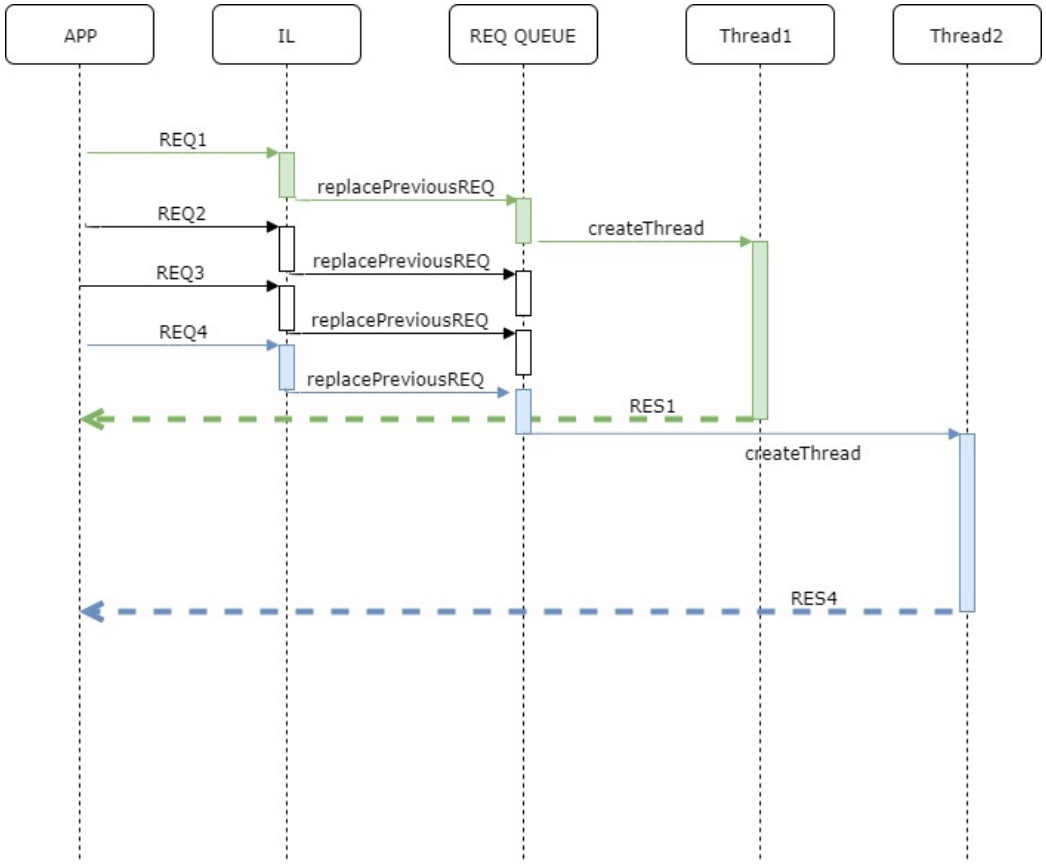

Figure 5.11: Sequence - ARAL

2. **App asks for the result when it is ready (AFR).** DIL will save the results in the response list and send a notification to the app. It will send the responses when the app asks for then. The time that the DIL save the response would be configurable to avoid poor memory usage by developers.

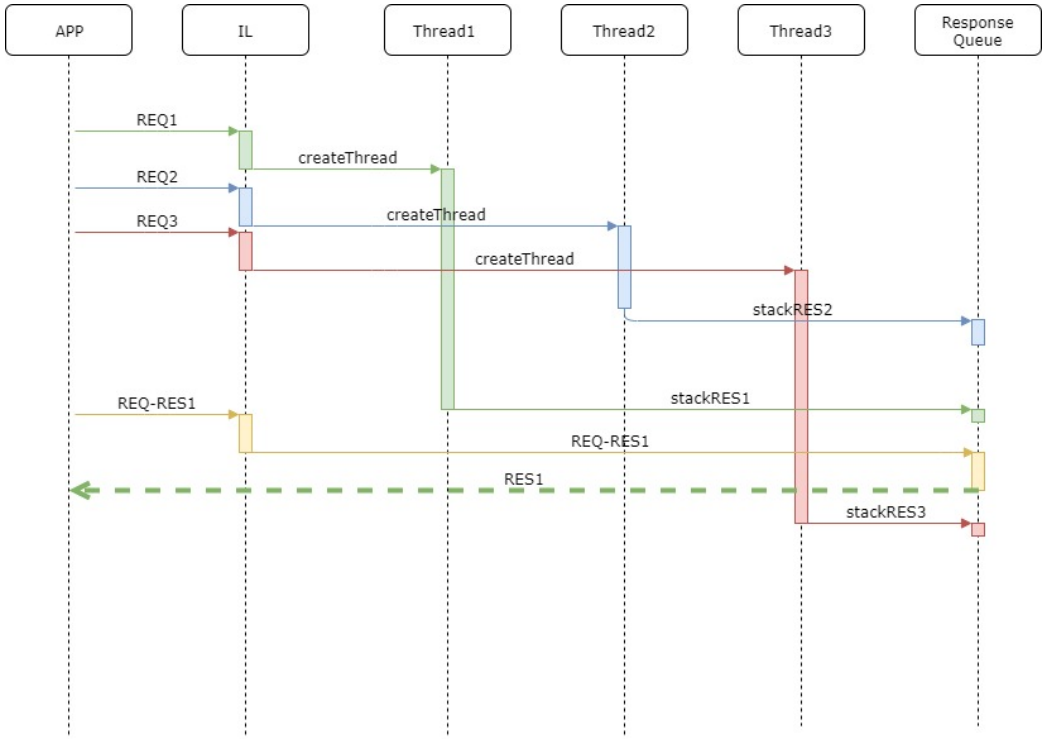

Figure 5.12: Sequence - AFR

This way to act is due to the large range of cases that exist and tries to adjust to the necessities of the developer of the applications. This purpose expects to give the tools to developers to use the Intelligence in the best way for the applications, therefore they are responsible to use and understand the DIL.

5.3.4 Availability

The large range of use case of the Intelligence Layer insight that it is not needed a fixed behaviour to execute or stop the DIL. The IoT devices are used for different proposes therefore, the witter propose a solution based on them. At the device starting, the DIL should act as its configuration:

- **Total availability**: The DIL would be always running as long as there are registered Apps, if not it will sleep. Thanks to this approach the DIL will be always ready for recalling ensuring a small response time. Suitable mode for streaming and single request.

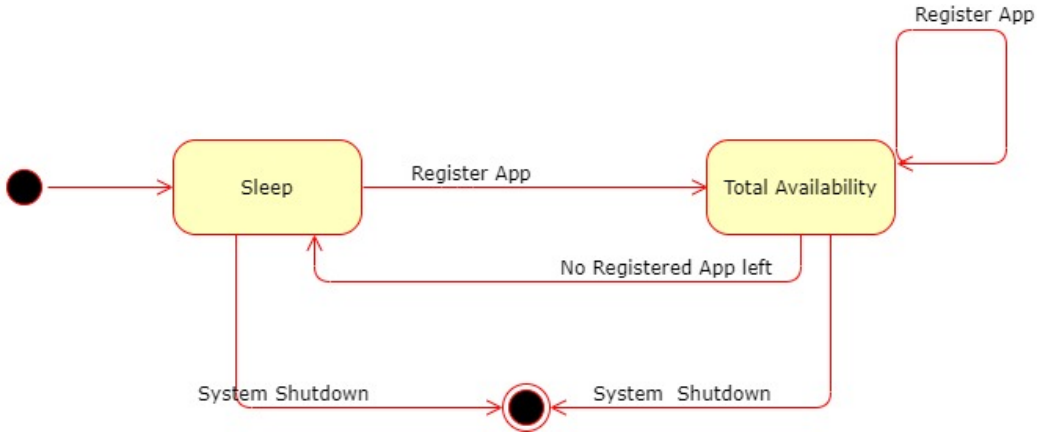

Figure 5.13: State Machine - Total availability

- **Partial availability**: Approach to reduce CPU consumption and perform single requests. When the DIL does not receive any request for a short time period it will sleep until the arriving of new requests no matter if there are registered apps or not.

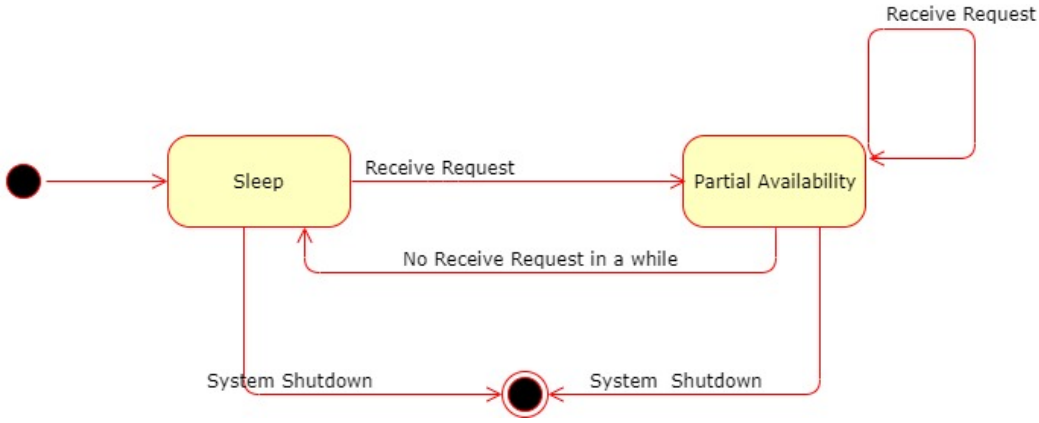

Figure 5.14: State Machine - Partial availability

	ARS	**ARAO**	**ARANO**	**ARAL**	**AFR**
Total	Small	Small	Small	Small	Small
Partial	High	Normal	Small	Hight	Normal

Table 5.1: Table of recommended behaviours

At sleeping, The DIL has to remain the input channels available to receive the register and recall requests, besides, it will save the registered Apps

information and the response list in a persistence way inside the device. Some approaches can be studied such as DIL users with different policies or cloud backup of the information but they are outside the scope of the thesis.

One approach to study in another thesis could be the total customization of the IS usage, in this way the recall case would be configured in a different way creating communication channels and personalizing the behaviour per each registration.

5.3.5 Apps Authentication in the DIL

In the scope of the Intelligence Layer, the Intelligence Services are resources that the Apps can request and use. Analyzing the role of the apps and the data involved, it is not a normal client-server case as it is not necessary a strong authentication phase for the Apps. Instead of authentication, the DIL needs an authorization protocol, therefore the DIL authorize the app to use the Intelligence Service. The protocol which suits more for this purpose is OAuth2 [30]. The idea is not to implement this protocol the idea is to adjust it to our necessities.

OAuth2 defines different roles:

- Resource Owner: the owner of the resource, in our case the resources are the Intelligence Services and the owner is the DIL.
- Resource Server: server hosting the resource. DIL hosts Intelligence Services.
- Authorization Server: server conducting access token to the client. The client will use the token to request the resource server. In our case, the DIL would generate the token for the Apps.
- Client: application requesting access to a resource server.

OAuth2 was made for different actors in different places meanwhile this thesis scope is inside one device, therefore, there are just two actors in the Intelligence Layer context, the DIL and Apps. When registering, DIL will generate the token for the Apps to allow them to use the Intelligence Services at recalling.

The token should be unique and confidential. One App can have multiple tokens to recall various Intelligence Services and various tokens can point to the same Intelligence Service.

5.4 DIL Architecture

The platform chosen to implement the Intelligence Layer was Android due to its extension, big community and the flexibility. It allows the perfect environment to grow and test this idea. This section will explore how to implement the Intelligence Layer in Android and the implementation decisions.

5.4.1 Implementation Decisions in Android

The first decision is where developing the DIL, it has to be accessible by all the applications as it is one requirement. The Android architecture and the Intelligence Layer concept aims to implement it in the Application Framework layer adding it as a new component but it implies that the solution would not be extensible to other platforms, therefore, the application layer is the desired option as all of the components can be reachable by the others.

Last versions on Android focus on reducing the background work[31] at minimum adding some limitations and encouraging to use threads. It does not fit the Intelligence Layer concept as it is needed an available agent to process the request, therefore the DIL should be developed as a Service inside an App.

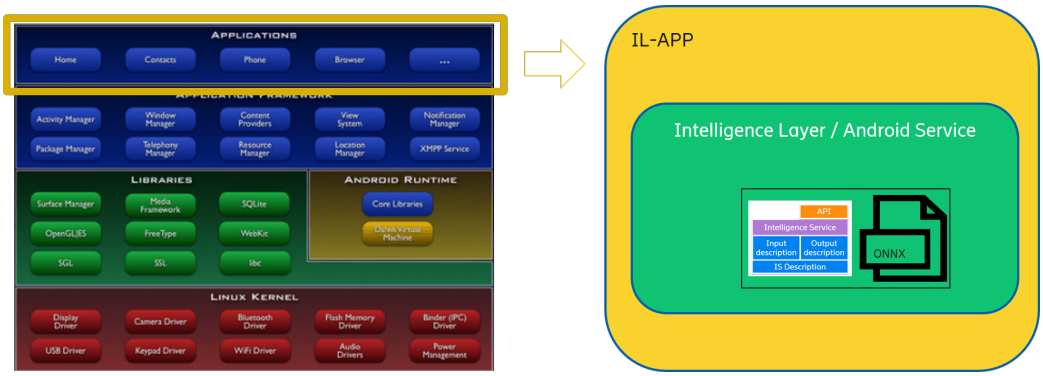

Figure 5.15: DIL as Android Service

Android offers different types of services to help developers to obey the background limitations; IntentService and IntentJobService, but their lifecycle and how they process the request are not customizable, therefore the normal Service is the chosen to develop the Intelligence Layer.

This service has to be developed in a way to obey the background limitations and the chosen availability and protocol. As Android is a platform with

huge user interaction and the response time has to be as small as possible, the availability is Total and for the example App, the protocol is ARANO.

The point-to-point channel cannot be implemented in Android, therefore, a two publish-subscribe channel patterns-called Broadcast in Android- is used in the implementation and the messages are encapsulated in Intents.

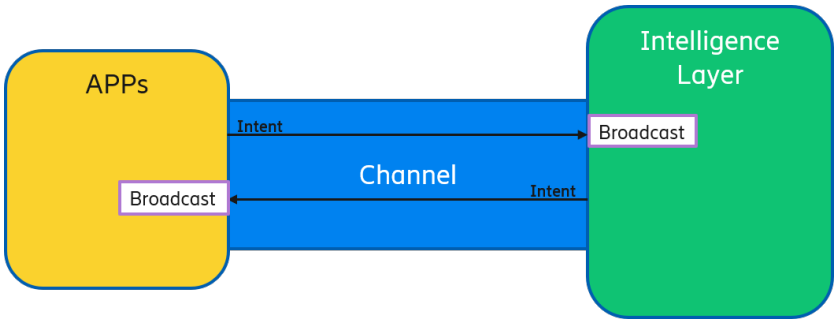

Figure 5.16: Android Channel

Android Services are another component in the system, therefore they can be called in different ways. To accomplish the background limitations and the desired availability the implementation use two ways:

- **startForegroundService()** function, this method is similar to the startService but it promises that the Service will call startForeground. In other words, it is sure that the component will call a background service which would be sent to the foreground. In the implementation StartForeground is called in the onCreate method, therefore as soon as the service is created it becomes a foreground service.

- **Broadcast**: It is an Android component similar to the publish-subscribe design pattern. These broadcasts are sent when an event of interest occurs. In this implementation, the DIL and each app have their own broadcast to exchange messages. The DIL broadcast is available for all the components of the OS and calls the startForegroundService to initialize the DIL Android service. At registering, the Apps have to share with the DIL their broadcast channel.

In order to make the DIL service available it is needed to modify the manifest file, adding permissions to execute a foreground service and publish the broadcast and the service:

Listing 5.3: DIL Manifest

```
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ericsson.client.intelligencelayer">

    <uses-permission
        android:name="android.permission.FOREGROUND_SERVICE" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"



        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category
                    android:name="android.intent.category.LAUNCHER"
                    />
            </intent-filter>
        </activity>
        <receiver android:name=".Service.customBroadcastReceiver">
            <intent-filter>
<action
    android:name="com.ericsson.client.intelligencelayer.Service.perform"
    />
            </intent-filter>
        </receiver>
        <service
            android:name=".Service.IntelligenceService"
            android:enabled="true"
            android:exported="true" />

    </application>

</manifest>
```

Note that the enabled and exported flags are true, therefore the service can be instantiated and executed by other components of the OS such as

Apps or services.

The way that the Android service handles the request is similar as explained in Figure 5.7. It contains a **ThreadPoolExecutor** which creates a new thread if the maximum thread is no reached, if not it queue the request. This implementation does not contain a responses list as the chosen protocol is ARANO and it is not needed.

Chapter 6

Evaluation

6.1 Intelligence Service Toolkit

The Semantic representation of the IS is really useful [20][19] as it allows to handle metadata easily and make queries without a defined structure focusing on the definition. On the contrary, it can be complicated for applications to process it due to the lightweight structure approach.

Therefore, the toolkit would be in charge to handle the raw output data, mapping to the Output Definition creating a deliverable Object for the Applications, such as a Class.

This Object has to be associated with the Semantic definition to keep the properties that the semantic offers, therefore technologies similar to JSON-LD or annotations in classes have to be incorporated.

The idea of this toolkit is to generate the Object/Class automatically at registering and include then in the toolkit library. Therefore the App and DIL can handle that information easily.

It can be incorporated in the Applications or in the DIL, the advantage to include it in the Apps is that they can use the semantic features easily and it reduces the DIL process time. Besides, to send an object among components it is needed to serialize it, therefore the schema has to be already in the DIL.

6.2 IS Management

The author realized that the token authorization needs some improvements as the initial approach was no consistent enough. Therefore, the author purpose an improvement including scopes to the token. The previous method failed when the DIL checked the input or output data as that information was lost in the registration procedure. For instance, using the previous approach the

DIL does not know the actual information the app wants to get, as the IS definition can contain multiple outputs meanwhile the app only is interested in one of them.

To avoid this the author proposes the use of scopes in the token. Scopes usually works to grant some permissions to the client but in this case, it will be used for the DIL to know what Input and Output are expected at recalling. The scopes would be useful for the toolkit and the DIL to know what output deliver to the Application in case the IS can give more than one.

This approach is also flexible as if there is a IS update, it reduces the changes in the registration and the app can use the same token, therefore just the IS definition changes but not the token. it is assumed that the changes in the definition do not affect to the IS main functionality, the changes would be related to the runtime or an Atomic Function with a better performance. In case of a IS with a different Input, Output or Purpose, it will be considered as a new IS and the registration process should be done again.

6.3 Improvements in Handling Requests

The proposed system has an architecture to handle requests. Nevertheless, it is not perfect as it presents issues caused by no error control and multi-threading. Therefore, the DIL should include some mechanism to avoid them, the author proposes a record of executing threads. This record would contain the information about the IS, the data source, the recipient apps, starting time and expected execution time.

- Errors. Whatever protocol is selected to communicate, the apps have to receive a response from the DIL even though performing the request produce an error. Therefore each thread has to be able to inform the app about the error, in addition, as there is a record of all of the performing threads, the DIL should send a notification if any thread overpass the expected execution time or if there is more number of treads record than executing threads.

- Multi-threading issues related to data management. The same data source can be requested to use for an IS, it can reach multi-threading issues. For example, the camera data can be used for different ISs. To avoid this problem the data source is divided into two types, the data outside the device is not included as it does not produce any multi-threading issues.

- Raw data: this data is inside the device and it can be found by a path. For example, it can be a picture, a fingerprint or a track of your positions. The DIL has to know all of them and does not execute a new thread if there is already another one using this data.

  There should be the possibility that the same raw data is used by different applications with the same IS, in this case, the DIL could perform just one inference and sending the result to both applications.

  To perform this, the DIL should record all of the thread information and the request queue to decide if a new thread can be created. When the thread finishes performing it will check the record list to know the response receivers.
- Sensor data: as its name denotes, it is data from the sensors, such as GPS or camera. The sensor availability is limited as just one application can use it at the same time, therefore they have to be managed by the applications to get the desired data to send to the DIL. This approach aims to reduce blocks produced by threads which are already using the sensor.

6.4 DIL Evaluation

The goal of this section is to evaluate the innovative idea of the DIL of decoupling the Intelligence from applications, comparing the results of the current state of the art, Intelligence belongs to the applications with the Intelligence Layer proposal. In order to analyze them, profiling was executed for each solution in an Android device, in particular, Samsung Tablet with this features:

- OS: Android 9.0
- CPU: QualcommÂ® SDM670 Octa-Core (2 x 2GHz + 6 x 1.7GHz) 14,2 GHz
- Memory: 4GB
- Storage: 64GB microSD card slot (up to 512GB)
- Battery: 7,040mAh

The profile is done recording the CPU and Memory usage and the execution time. The graphs can be confusing, therefore the author includes a brief explanation of them.

- CPU timeline:
  - Red dot. A touching event in the device, the inference of IS is done after this event.
  - Green trace, CPU usage by the APP.
  - Blue trace, CPU usage by OS and other apps.
  - Thread activity timeline. Information about the active threads and their execution.
- Memory:
  - Java: Memory from Java or Kotlin objects.
  - Native: Memory from C or C++ code.
  - Graphics: Memory used for graphics buffer queues to display pixels to the screen, including GL surfaces, GL textures, and so on. (Note that this memory is shared with the CPU, not dedicated GPU memory.)
  - Stack: Memory used by both native and Java stacks in the app.
  - Code: Memory that the app uses for code and resources, such as dex bytecode, optimized or compiled dex code, .so libraries, and fonts.
  - Others: Memory used by the app that the system isn't sure how to categorize.
  - Allocated: The number of Java/Kotlin objects allocated by the app. This does not count objects allocated in C or C++.

To start this analysis, the result of the state of the art approach -app and Intelligence coupled- is shown:

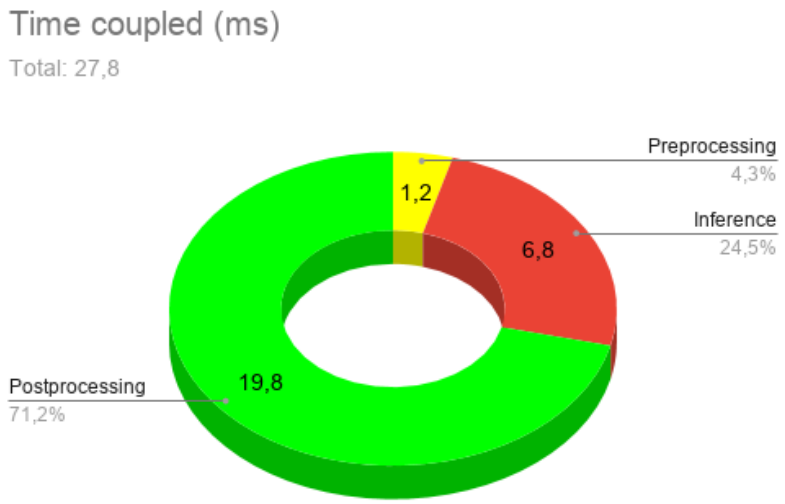

Figure 6.1: Coupled Intelligence Execution time

As expected, the processing time is low, in the processing part is included the time to show the result in the screen, therefore is higher than just the normal postprocessing of the machine learning algorithm.

The following figure shows that during all the inferences the CPU usage presents spikes at inference

Figure 6.2: Coupled Intelligence - CPU

The values are about 5-7% as shown in the next picture:

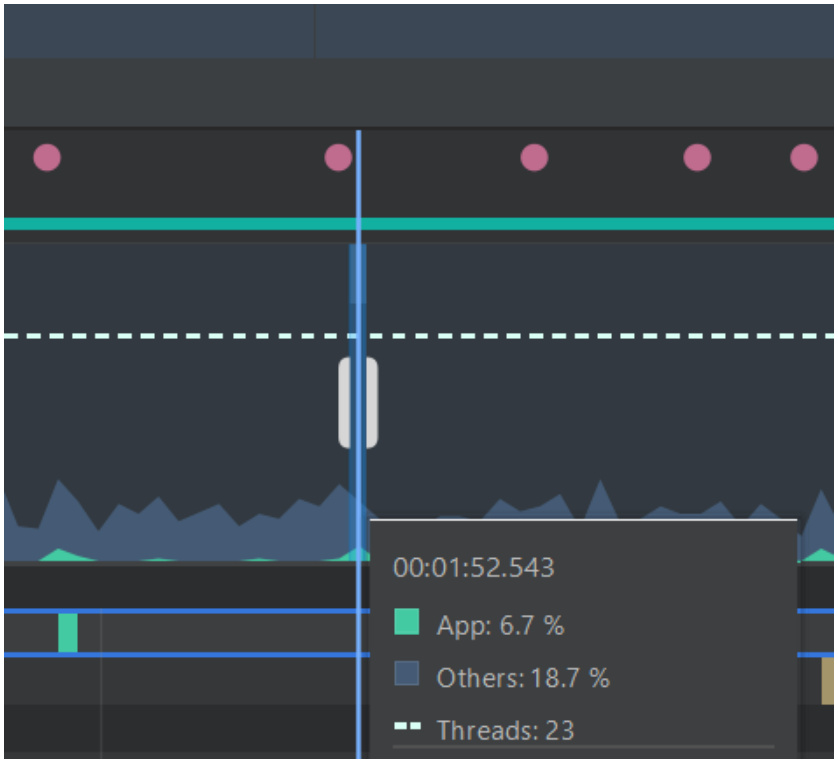

Figure 6.3: Coupled Intelligence - CPU

The memory graph only shows a big difference in graphics memory. It is due to a change in the view.

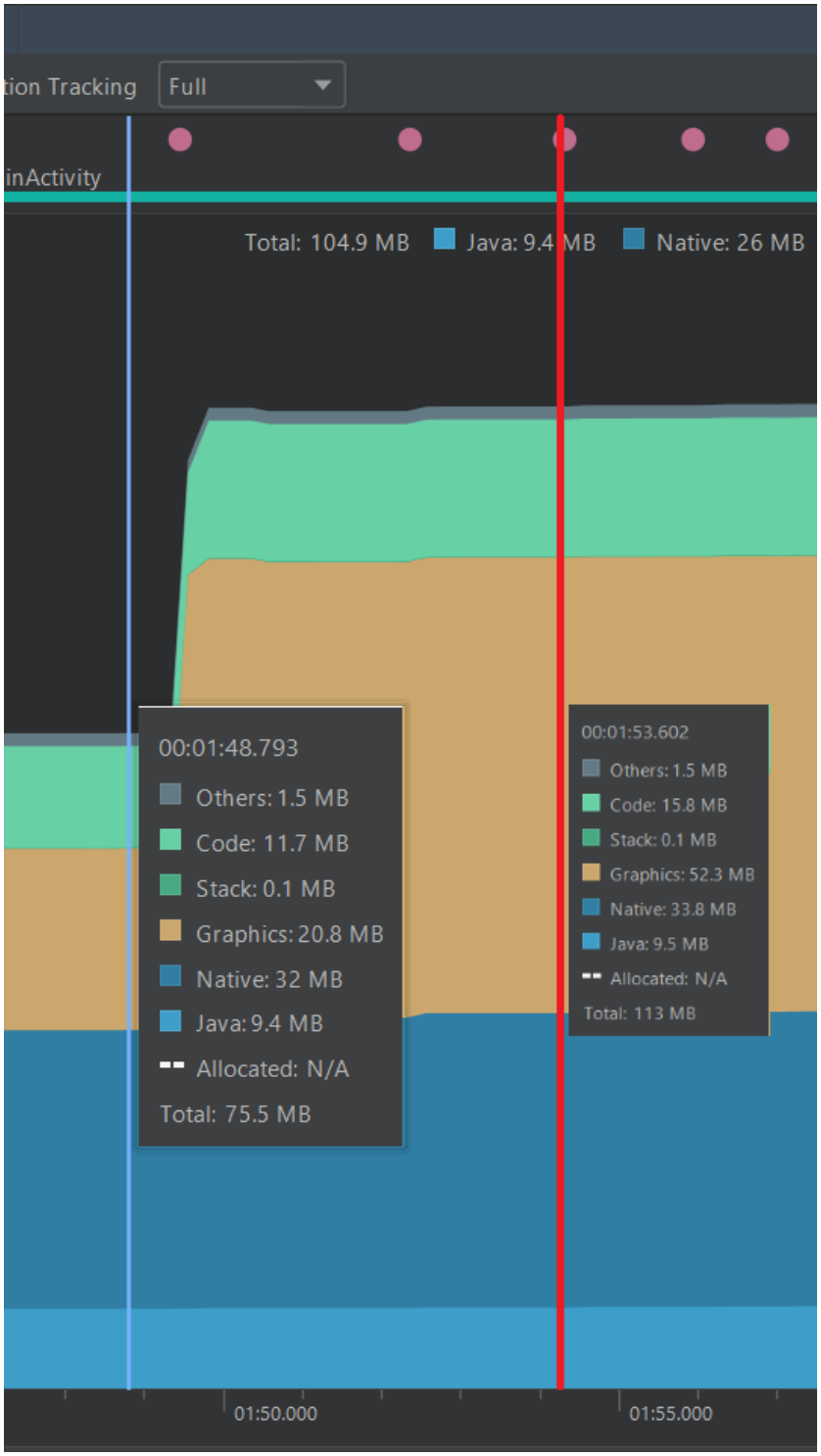

Figure 6.4: Coupled Intelligence - CPU

When analyzing the implemented service, the tests were not consistent due to the high difference of the performance, some times the process to receive a response from the DIL and process it takes about 650 ms an other times it takes 120 ms. The only consistent characteristic in these tests is the percentage of execution for each function, therefore, the analysis will focus on this feature to analyze the implementation.

This figure shows the time of execution for each actor, the DIL (server) and the app:

Inference Decoupled

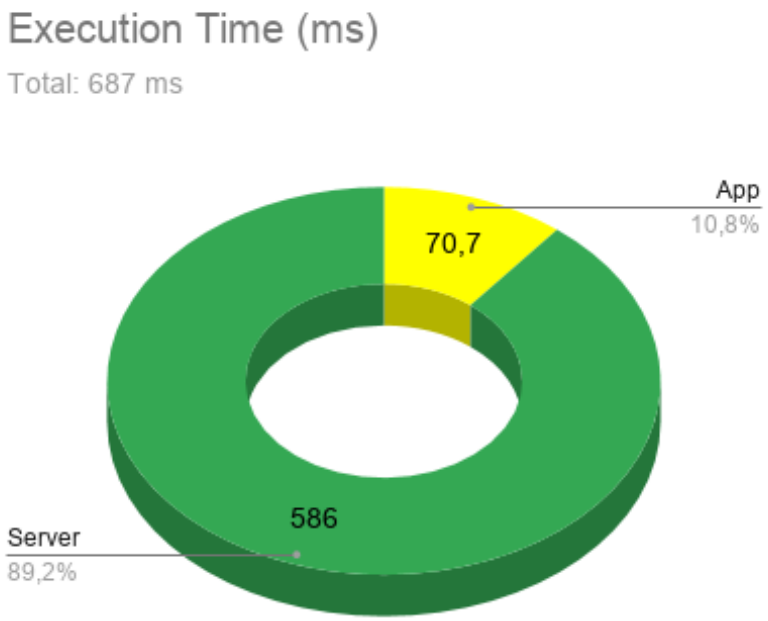

Figure 6.5: Execution time

As expected, the time of the server processing is higher than the app processing, in particular it is almost nine times higher.

The processing time of the app is mainly used by the toolkit, the "Other" refers to OS time processing, such as showing information or message notification.

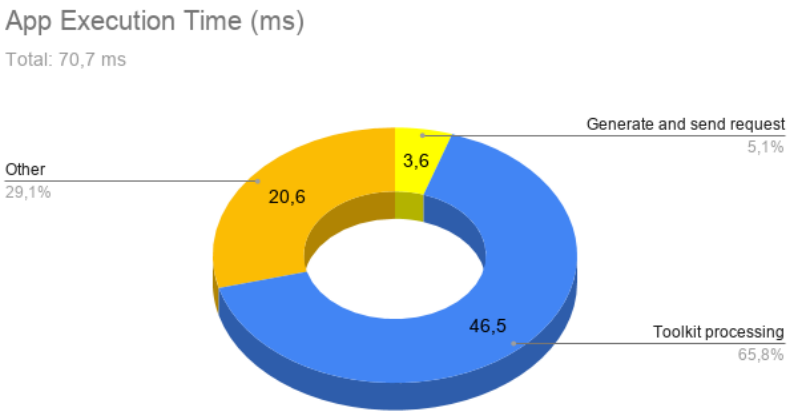

Figure 6.6: App Execution time

On the other hand, the inference of the IS represents most of the time execution.

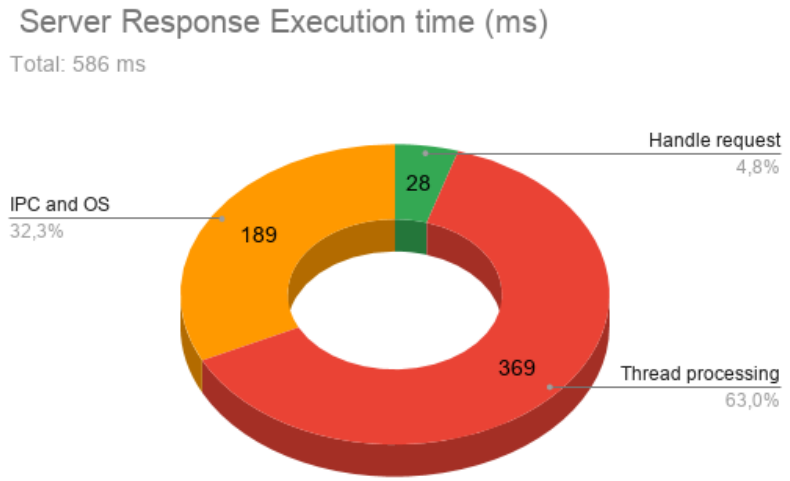

Figure 6.7: Server Response Execution time

At performing the IS, the actual inference of the Intelligence is not relevant, to understand the IS API represent the 80% of the tread execution and almost the 45% of the total process time.

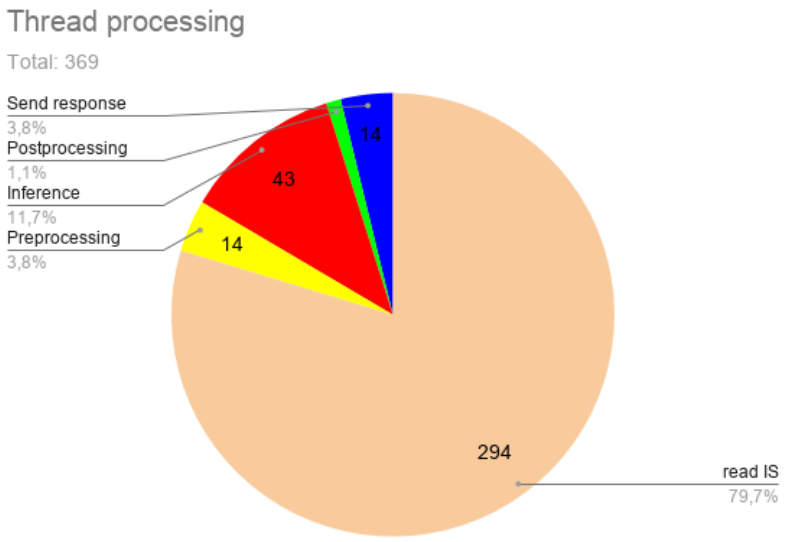

Figure 6.8: Thread Response Execution time

The CPU usage during the execution varies from 35-50%, 1% to 5% is from the app and the 35% to 45% is related to the service and OS. App side figures:

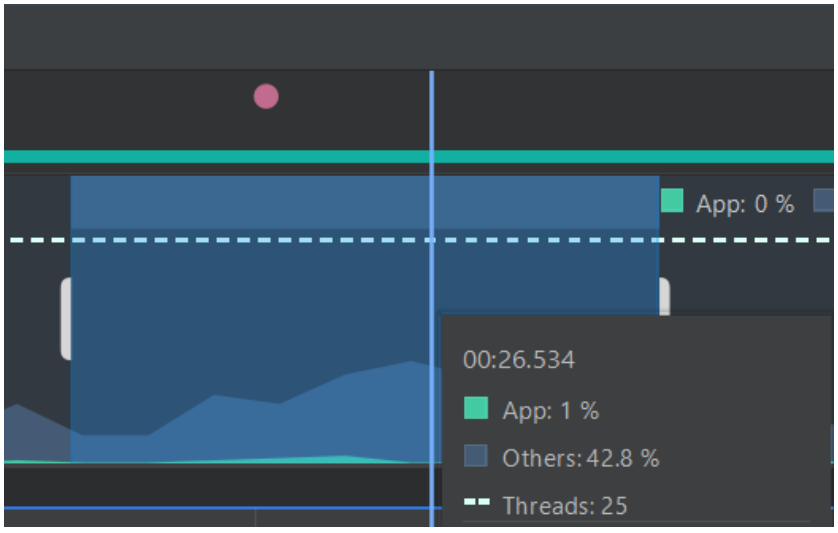

Figure 6.9: CPU Usage APP - Message delivered to the server

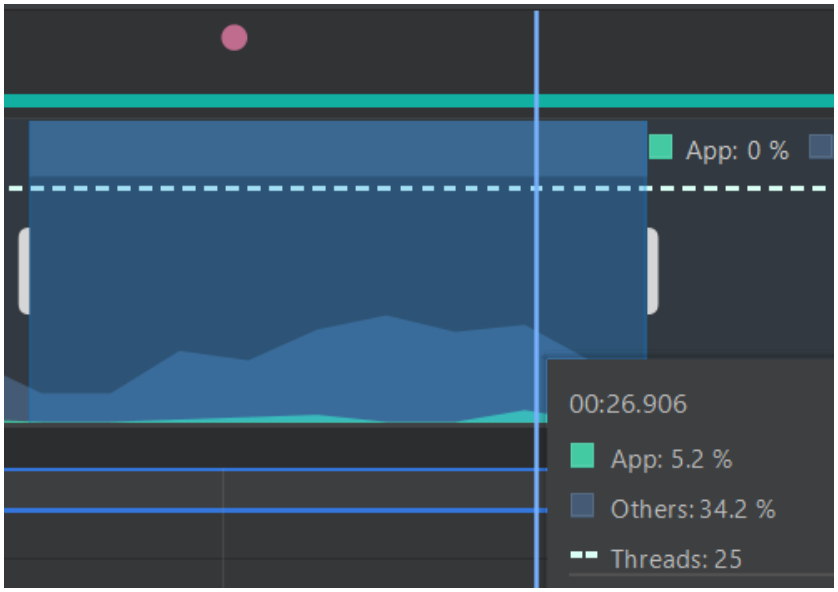

Figure 6.10: CPU Usage App - Response from the server

That figures show the information from the application side and do not give proper information about the service, therefore, separate profiling was done on the server side, performing twenty inferences.

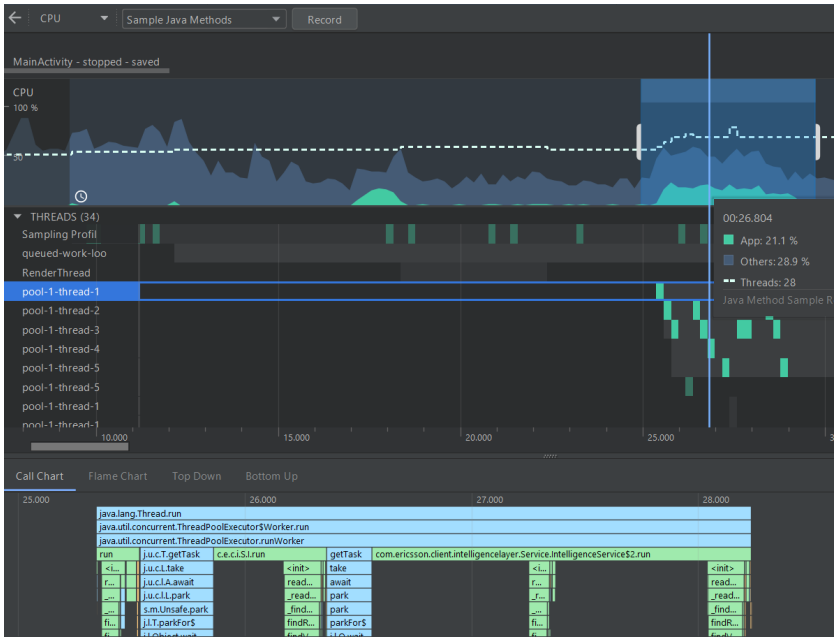

Figure 6.11: CPU Usage Service - Handling multiple Request

The server CPU usage is about 20% of a total of 50%. The values are similar to the previous values, therefore the author concludes that the average CPU usage at inference is:

- Client: 5%
- Service: 30%
- OS: 15%

Comparison figure of the both approaches:

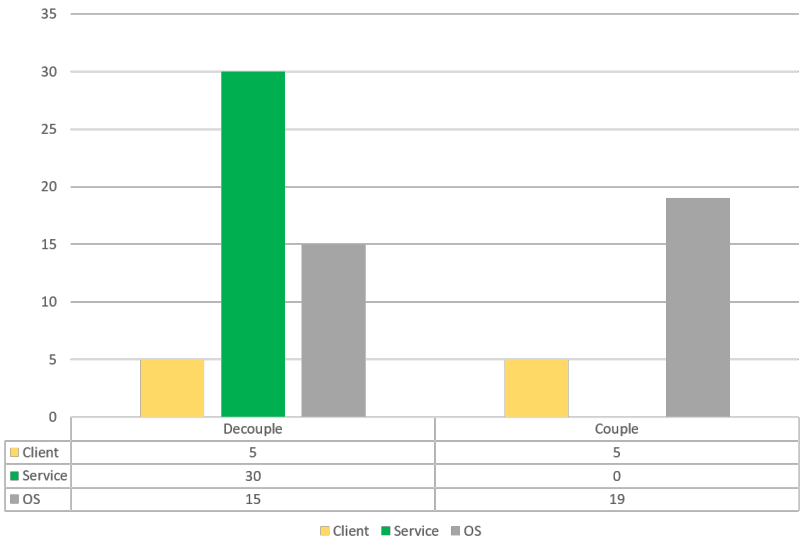

Figure 6.12: CPU Usage Comparison

*Note that during the tests, when the execution time was really low -about 120 ms-, the CPU usage levels were lower but the author is not sure if they were low enough to reduce the execution time by six times.

On the other hand, the memory usage does not change even though the performance time do. Actually, it behaves as the state of the art app, the memory remains the same in the process and the only change is in the graphs memory due to the notifications and changes of the views.

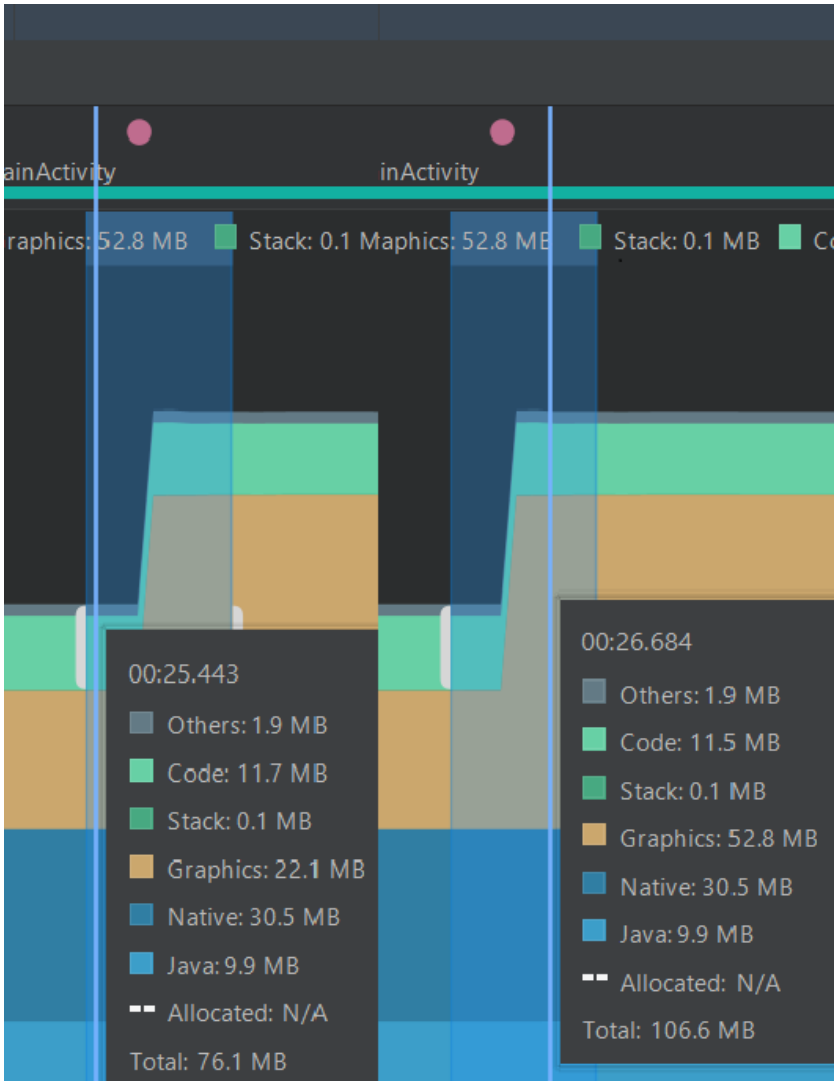

Figure 6.13: Memory before and after the inference

Chapter 7

Discussion

The current implementation allows applications to register and inference multiple ISs as well as customizing the communication behaviour in order to use one of the general protocols proposed AFR or AR.

The first state of the thesis had some issues related with the internal control of requests at recalling and registering due to the multi-threading approach, in addition, there was a lack of information in the app side. In order to avoid these issues, the author purposed some elements: a toolkit requests control and improvement of the token usage. The current implementation does not perform all of them as the main objective is to test the decoupling of Intelligence. It contains the toolkit and little multi-threading control, anyway the test performed as it was expected to show that decoupling Intelligence is reachable.

Nevertheless, the implementation is not consistent as there is a big difference in the execution times at performing the IS inference. The author believes that it is due to the Android OS as the time percentage of each function remains for different executions. In order to solve this, it would be necessary a deep experience in Android but unfortunately, the author has not this experience.

Anyway, the results were clear, it is possible to decouple the Intelligence from applications and orchestrate it. This is possible due to different factors:

- Intelligence definition.
- Discovery.
- Registering.
- Recall.

The results also showed some issues in the implementation, the execution time of the IS API understanding is about 45% of the total time. Therefore it is needed a better representation, to solve this other formats can be studied such as XML, RDF or Protocol Buffers. Protocol buffers are simpler and faster than XML/JSON [32][33][34], this features become protocol buffers a good choice to research in the future, in addiction this technology support versioning and flexibility in the structure, moreover the ".proto" file introduce some security in the interaction as protocol buffer is only meaningful with this file.

Due to the lack of time, it was not possible to make experiments with the different protocols proposed in this thesis, but the author believes that they will be useful in the future or they can be the basic ones for a better communication paradigm. Besides, the token and the multi-data-source approached should be tested. As there is a lack of IS implementations, they could not be fully tested. Still, these are the initial approaches and there can happen that they do not work for some situations, therefore the author highly recommends to develop more ISs from different scopes and data source to test these thesis proposals.

Chapter 8

Conclusions

This thesis proposes a new paradigm offering Intelligence as an on-device service. The result was clear and it can be achieved. There are already some attempts on providing Intelligence as a service but all of them failed due to different factors, such as restrictive environment, lack of consensus or no profitable options. The Intelligence Layer offers all of them, the ecosystem provides the perfect environment for companies to get profit at sharing knowledge and the semantics definition allows all of the actors to achieve consensus.

The conceptual stage of the Intelligence Layer makes it a challenging project and made a big impact on the results and conclusions as the implementation had to be developed almost from scratch. In addition, although the project counted with big conceptual support, most of the features were not settled and they had to be discussed. However, the author believes that this project can revolutionize the current state of the B2C and B2B industry.

The IoT context of this project also took a big part in making decisions. The diversity of IoT devices encouraged the author to research for flexible solutions that can be implemented for multiple contexts. Therefore, the author reached the conclusion that the flexibility of the DIL takes high importance and he purposed different communication protocols based on how the applications want to receive the responses. In addition, the author developed different DIL life cycles depending on the expected DIL usage. The research also focuses on the authentication of the apps, as the processes performed in the device, a strong authorization is not needed, purposing an authorization approach based on OAuth2.

The main goal of the thesis is to demonstrate that the decoupling if the Intelligence in an application can be achieved and research how this innovative approach should be done focusing on the registration and inference of the Intelligence. Therefore the performed tests focused on these tasks.

The current state of the solution suffers a big limitation, the number of ISs is limited affecting the test. Nevertheless, the DIL performs as it is expected for the cases of registration and recall. As explained in the DIL use cases, the DIL has to perform discovery, registration and management of policy, dependencies and authorization. The solution proposed offers all of them without discovery, policies management and dependencies management.

The results clearly showed that Intelligence can be decoupled from the applications. Nevertheless, they also showed a big unstable performance, the executing time when inference varies a lot but as the percentage of time remains the same no matter the execution time the author believes that this change is due to the background task performing in Android without control. An API to inference the ISs based on JSON is proposed, processing this API takes about the 45% of all the execution time for a lightweight model, therefore it will be interesting to find another way to represent the API, such as protocol buffers as they perform faster[32][33][34]. Comparing the state of the art application with the Intelligence-less application, the coupled one performs faster, with a similar usage of CPU and memory of the decoupled one. The execution time is not a proper factor due to the unstable circumstances but the decoupled application used the DIL to register to recall a different Intelligence within seconds. Meanwhile, the development of the new functionality in the coupled application would take some hours and would not be reusable for other applications, therefore the author concludes that the research in the Intelligence Layer must have a high priority for the industry as it provides a way to reduce cost and one approach which can be reusable for multiple domains.

Due to the flexibility of this approach, the author proposes that a research with different use cases of Intelligence would be beneficial to find new behaviours and improve the solutions proposed in this thesis. In addition, new technologies can be explored such as JSON-LD, XML or protocol buffer for defining the IS API and another OS to implement the Intelligence Layer to analyze if this architecture performs properly in another device. Besides, defining more IS and analyzing the best way to utilize them can be a key factor when inference, it can show an insight into an API representation improvement or more suitable communication protocols. The registration case can be analyzed as well to find more needs, such as new usage of the token scopes or a totally new approach. The author conclusion is that the level of the Intelligence Layer is mostly conceptual and this thesis is the first implementation of the idea, therefore, it is needed to be tested in a real case to get real insights of what can be improved due to the magnitude of this project.

Bibliography

[1] Edgar Ramos, Roberto Morabito, and Jani-Pekka Kainulainen. Distributing intelligence to the edge and beyond. https://www.researchgate.net/publication/329183219_Distributing_Intelligence_to_the_Edge_and_Beyond, 11 2018.

[2] ONNX Model Zoo. https://github.com/onnx/models. Accessed: 2020-03-03.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf, 2016.

[4] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. http://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf, 2011.

[5] M. Kalal M. Kumar R. Membrilla A. Siviero Y. Torisu N. Wallis C. MacGillivray, M. Torchia and S. Chaturvedi. Worldwide internet of things forecast update, 2016-2020, 2016. IDC Research.

[6] Gurpreet Kaur and Mohammad Muztaba Fuad. An evaluation of protocol buffer. https://www.researchgate.net/publication/261336190_An_evaluation_of_Protocol_Buffer, 2010.

[7] Protocol Buffers Reference. https://developers.google.com/protocol-buffers/docs/reference/overview. Accessed: 2020-03-03.

[8] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml) 1.0. https://www.w3.org/TR/xml/, 2008.

[9] ONNX GitHub page. https://github.com/onnx/onnx/tree/master/onnx. Accessed: 2020-03-03.

[10] Official release of ONNX Runtime. https://azure.microsoft.com/en-us/blog/onnx-runtime-is-now-open-source/. Accessed: 2020-03-03.

[11] JAVA. https://go.java/?intcmp=gojava-banner-java-com. Accessed: 2020-03-03.

[12] Java Navie Interface - JNI. https://docs.oracle.com/en/java/javase/11/docs/specs/jni/intro.html#java-native-interface-overview. Accessed: 2020-03-03.

[13] Inter Process Comunication - IPC. https://www.geeksforgeeks.org/inter-process-communication-ipc/. Accessed: 2020-03-03.

[14] Message passing vs Shared-memory - IPC. https://www.tutorialspoint.com/message-passing-vs-shared-memory-process-communication-models. Accessed: 2020-03-03.

[15] Message passing patterns. https://www.enterpriseintegrationpatterns.com/patterns/messaging/. Accessed: 2020-03-03.

[16] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Shared memory vs message passing. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.4066, 2003.

[17] Android. https://developer.android.com/. Accessed: 2020-03-03.

[18] Android Services. https://developer.android.com/guide/components/services. Accessed: 2020-03-03.

[19] Advantages of semantic data. http://www.semagix.com/advantage-of-semantic-data.htm. Accessed: 2020-03-03.

[20] Jennifer Rodd and Gareth Gaskell. The advantages and disadvantages of semantic ambiguity. https://www.researchgate.net/publication/2403793_The_Advantages_and_Disadvantages_of_Semantic_Ambiguity, 07 2000.

[21] Resource Description Framework - RDF. https://www.w3.org/RDF/. Accessed: 2020-03-03.

[22] Web Ontology Language - OWL. https://www.w3.org/OWL/. Accessed: 2020-03-03.

[23] Advantages of RDF. https://carbonldp.com/blog/2017/12/18/the-advantages-of-resource-description-framework-rdf/. Accessed: 2020-03-03.

[24] SPARQL. https://www.w3.org/TR/rdf-sparql-query/. Accessed: 2020-03-03.

[25] RDF to JSON. https://www.w3.org/TR/rdf-json/. Accessed: 2020-03-03.

[26] Rdf to json-ld.

[27] Martin Ledvinka and Petr Kremen. A comparison of object-triple mapping libraries. http://semantic-web-journal.net/content/comparison-object-triple-mapping-libraries, 12 2018.

[28] Point to Point Channel. https://www.enterpriseintegrationpatterns.com/patterns/messaging/PointToPointChannel.html. Accessed: 2020-03-03.

[29] Request-Reply pattern. https://www.enterpriseintegrationpatterns.com/patterns/messaging/RequestReply.html. Accessed: 2020-03-03.

[30] OAuth 2.0. https://oauth.net/2/. Accessed: 2020-03-03.

[31] Android Execution Limitations. https://developer.android.com/about/versions/oreo/background. Accessed: 2020-03-03.

[32] Protocol buffers vs JSON . https://www.bizety.com/2018/11/12/protocol-buffers-vs-json/. Accessed: 2020-03-03.

[33] Beating JSON performance with protobuf. https://auth0.com/blog/beating-json-performance-with-protobuf/. Accessed: 2020-03-03.

[34] Choose Protocol buffers. https://codeclimate.com/blog/choose-protocol-buffers/. Accessed: 2020-03-03.